# Improving the Capabilities of JFLAP

## Creating Effective User Interfaces in Learning for Theoretical Computer Science

**Ian C. McMahon**
Computer Science Department
Duke University
Durham, NC 27708
icm9@duke.edu

**Abstract**: JFLAP, the Java Formal Languages and Automata Package, is a software visualization tool for teaching various topics in theoretical computer science, including formal languages theory, grammars, and automata. Consisting of elements to represent formal languages of different classes (including regular, context-free, and recursively-enumerable), types of automata (including nondeterministic finite automata, pushdown automata, and Turing) machines, grammars, regular expressions, context-free and regular pumping lemmas, and Lindenmayer Systems (L-Systems), JFLAP is used internationally in various courses at the college level. The current version available to the public, JFLAP v7.0, is restrictive in its static codebase, not allowing for easy modification or additions, conceals the underlying formal definitions, and limits symbol naming conventions. To improve upon JFLAP's capabilities, much of the previous code has been redesigned, focusing on new class-based hierarchies to extend its flexibility, improving upon existing concepts to make the graphic user interface easier to interact with, and adding new functionality to broaden its capabilities. This reconstruction and extension has dealt with many of the issues facing the old version of JFLAP while maintaining similarity to avoid user confusion. To compare the two versions, an assessment survey was conducted regarding some of the new and modified capabilities. The survey indicated that the changes improve upon the old version and expand JFLAP's pedagogical capabilities.

This work was supported by the National Science Foundation through NSF grant DUE-1044191

# 1. Introduction

Automata and formal languages theory is an important and prevalent subject in computer science which includes concepts such as grammars, automata, and formal languages. It provides the foundation of theoretical computer science, upon which topics such as compilers, programming languages, discrete mathematics, and graph theory are built upon. Due to its theoretical nature, it is often very difficult to receive immediate feedback in problem solving in this area, but has been aided over the past 50 years with software and algorithm visualization tools [CSK 2011, Lin 2006].

In the past, automata and formal languages theory has been taught through the use of formal definitions and pen-and-paper representation of the abstract topics, algorithms, and proofs. These methods, while providing the building blocks of computer science as we know today, are tedious and difficult to both teach and learn. Many of the topics require trial-and-error or manipulation of the concrete representations, and are prone to human error. To represent the topics as handwritten tuples or hand drawn graphics at each step of their operation is clearly tedious. A platform with the ability to symbolize, interact with, and reuse these concepts was therefore sought after, and was supplied by the emerging fields of software and algorithm visualization.

Software visualization is defined as “the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction and computer graphics technology to facilitate both the human understanding and effective use of computer software” [PBS 1993]. Much of the research on software visualization has focused around levels of user engagement, factors in the pedagogical effects of animations, effectiveness of visualizations depending on knowledge or comprehension level of topics, and the characteristics of visualizations that prove to be effective in increasing learning capacity [SBL 1993, BCS 1999, HDS 2002, Nap 2002, SSMN 2004].

As early as 1963, researchers have been implementing software visualization programs to provide a medium through which a user can interact with automata and formal language theory. Many tools have been created since then, most of them focusing on a specific sub-topic of

automata and formal languages theory [SCH 1994], though with some covering multiple topics [Tay 1998]. Some tools were designed around programming languages similar to the original tuple representation of the topics, while others allowed for direct manipulation of graphical elements that emerged from the paper-and-pen representations. Other approaches have included the *hypertextbook*, a web-based teaching/learning resource, with aims of providing a common layout with incorporation of multimedia technologies, cross browser viability, and simpler deployment [Ros 2008]. The number of platforms makes it difficult for there to be an all-encompassing, comprehensive software visualization tool for automata and formal language theory, which was part of the inspiration of the JFLAP project.

JFLAP, the Java Formal Language and Automata Package [Rod 2013, RoF 2006], is one of the world's most sophisticated software visualization tools for teaching various topics in theoretical computer science due to two decades of development [CSK 2011]. People from over 160 countries have downloaded JFLAP. With its graphic visualization tools to aid in the learning of formal languages, finite automata, Turing machines, grammars, parsing, and L-systems, JFLAP provides for a comprehensive framework.

Started under Professor Susan Rodger at Rensselaer Polytechnic Institute in the early 1990s as a tool for visualizing Nondeterministic Pushdown Automata, JFLAP was initially called "FLAP," and was written in C++ and X-Windows. As it was developed over the years, it integrated more topics, starting with different automata (finite automata and Turing machines), then grammars, parsing, and conversion algorithms from grammar to automata and vice-versa. After coming to Duke University, Professor Rodger continued work on the project, adding more grammar parsing algorithms, Moore and Mealy machines, Lindenmayer Systems, and pumping lemma components, transitioning the code to Java (where it was renamed JFLAP), and co-authoring a user manual with student Thomas Finley centered around JFLAP [Gen 2012]. More recently, JFLAP has been expanded further, with hopes of writing a textbook and building an entire automata and formal language theory curriculum around it, with the additions of batch testing, image exporting, and magnification. A tutorial page at www.jflap.org gives anyone access to explanations on the entire program's components, as well as sample files and the program itself to download.

However, one of JFLAP's limitations is the fact that much of the underlying code is quite static, and developed separately from the rest of the package, which does not allow for easy modification or additions, especially when it comes to the idea of the underlying symbols that are paramount in the creation, manipulation, and learning of formal languages.

A formal language is defined as a subset of ∑*, where ∑ is an alphabet of symbols and ∑* is words formed from all possible combinations of zero or more symbols from ∑. Thus, more colloquially, a formal language is a set of words over an alphabet. There may be rules or constraints on the language to specify a certain layout in the formulation of these words. What is significant about this definition in visualization, and where previous versions of JFLAP have had a problem, is the fact that these symbols are limited to how the piece of software interprets a "symbol." In JFLAP v7.0 [RQS 2011], as in its predecessors, each symbol was represented as a Java String, and essentially limited the available choices to single alpha-numeric characters(sometimes including special symbols such as λ, (, +, &, %, etc.), where upper-case letters were used solely for variables and lower-case for terminals (which are central to grammars and will be explained later).

While there is nothing inherently wrong with this representation, it ruled out the ability to create complex languages that could actually represent structures and natural languages we see in real life (for example, variables called *Subject*, *Object*, *Verb*, and terminals representing real words). Along with this came the fact that some of the underlying code revolved around String manipulation, which could cause errors if the user wanted to select symbols that the manipulation utilized for other purposes (for example '/' in some parsing algorithms), and that would cause the program to malfunction. The solution was to disallow specific characters, which will have to be done occasionally in the new version, but it was certainly not a lasting fix in the terms of flexibility. Changing the JFLAP representation of symbols was not something that could be done in one place to fix the problem. Symbols are the building blocks of just about every part of the program, and to allow for flexible alphabets while preserving all of JFLAP's features would affect the entire package. This decision resulted in a complete code redesign, which began to incorporate these new *Symbols* and *SymbolStrings* of multiple *Symbols* [Gen 2012]. It also led

to a reconstruction of the codebase to follow a more Object-Oriented framework and meet newer coding practices, such as the use of generics, to provide a JFLAP that would allow for easier adaptation in the future and not only be useful as a teaching/learning visualization tool on its own, but also more modifiable by users.

Along with the redesign and general code cleanup, there were some algorithms JFLAP included that were not fully clarified or visualized, or that had room for some significant improvements. Our work has included changes to the JFLAP program in the form of CYK parsing, Brute Force Parsing, Turing machines (TMs) including Universal Turing Machines and building block-TMs, some conversion algorithms, implementation of a graphic user interface, modification of the package's hierarchy, and language "generation". The work on each has laid a framework that will add significant capabilities to JFLAP, and improve many of the aspects of the visualization software, including its capabilities as an educational tool.

# 2. Current Functionality

The current version of JFLAP available to the public, version 7.0, is composed of eleven environments that are presented to the user upon starting the program. A brief overview of each is given below.

One of the more important aspects of the software visualization is a logical representation of the various components, automata most of all. While grammars, regular expressions, etc. can be more or less visually represented by fairly plain text, automata are complex "machines" with states and transitions. JFLAP uses a transition graph representation of automata, in which states are vertices and transitions are directed edges within the graph [Lin 2006]. States possess *names*, which are unique identifiers each state is presented with, but may also have a *label* for more information. Each transition is given a label, depending on what symbols it uses, and that information is presented on the directed edge. The start state is represented by a large triangle attached to the left of the vertex, while final states (which are present in acceptors, but not transducers) are represented with a second, smaller circle within the vertex.

Commencing with the start state, an input string is parsed, taking transitions from state to state depending on the current location in the string and graph. If the entire string is parsed, and the machine ends in a final state (or the machine is a transducer), then that string is accepted. Similar to a hand drawn machine, though with much more flexibility (including undoing, redoing, deleting, and manipulation), the user can create a machine with any number of states and transitions, and can add labels and notes as a way of marking states or writing reminders of functionality. Transitions can be edited after creation, and the layout of the graph is dynamic, allowing for states to move, transitions to be curved differently, and states to be renamed.

## 2.1 Finite Automaton

The formal definition [Rod 2013] of a finite automaton (FA) $M$ is the quintuple $M = (Q, \sum, \delta, q_0, F)$ where:

| |
|---|
| $Q$ is a finite set of states, $\{q_i \mid i$ is a nonnegative integer$\}$ |
| $\sum$ is a finite input alphabet of symbols |
| $\delta$ is the transition function, $\delta$: $Q \times \sum \rightarrow Q$ |
| $q_0 \in Q$ is the start state |
| $F \subseteq Q$ is a set of final states |
| **For non-deterministic FAs, $\delta$: $Q \times (\sum \cup \{\lambda\}) \rightarrow 2^Q$ |

This definition includes both deterministic and non-deterministic finite automata (DFA and NFA respectively). Both are classified as an acceptor, that is, they will either accept or reject a string. DFAs accept a string $w$, transitioning through states by scanning the string from left to right, one symbol at a time. The machine is deterministic, so at any given point, there is only one transition given a specific symbol, and there are no $\lambda$-transitions. If $|w| = n$, the runtime of a DFA accepting or rejecting a string is worst-case $O(n)$. The language accepted by a DFA is denoted $L(M)$ and is a regular language.

NFAs are slightly different from DFAs in that the transition function allows for $\lambda$ (that is, the empty symbol) as a transition symbol, and its range is the powerset of $Q$. Put in simpler terms, non-determinism allows for multiple transitions out of a state on the same symbol, allowing for the machine to go into different states with the same input. However, this does not imply they are

more powerful than DFAs. In fact, all NFAs can be converted to an equivalent DFA, and therefore, the language accepted by an NFA is also regular.

There are many algorithms provided by JFLAP v7.0 involving FAs, including stepping through input, converting NFA to DFA, testing for equivalence of two FAs, combining automata, adding trap states, minimizing DFAs, and converting to a regular grammar.

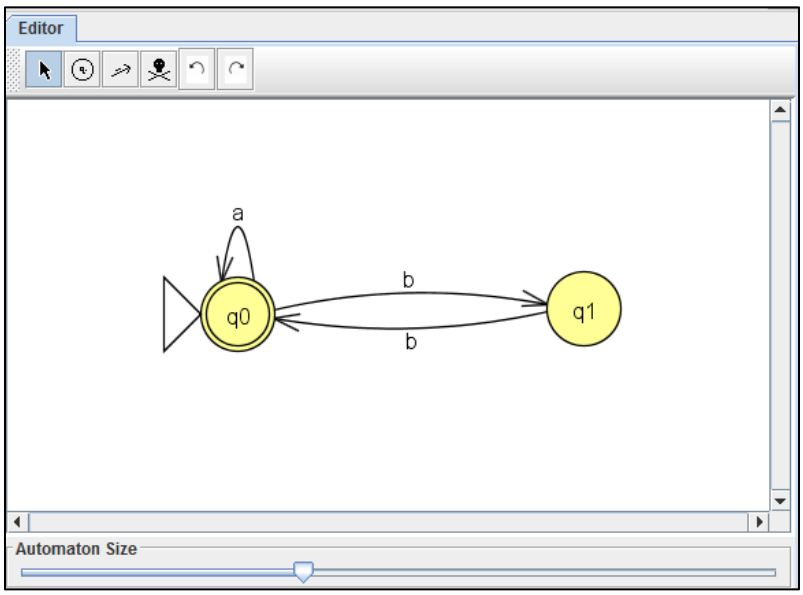


Figure 1: A basic Finite Automaton in JFLAP v7.0

## 2.2 Mealy Machine

A Mealy machine $M$ is similar to a FA, but with a few select differences. The machine is defined by the sextuple $M = (Q, \sum, \Gamma, \delta, \omega, q_0)$ where:

| |
|---|
| $Q$ is a finite set of states |
| $\sum$ is a finite input alphabet |
| $\Gamma$ is a finite output alphabet |
| $\delta$ is the transition function, $\delta$: $Q \times \sum \rightarrow Q$ |
| $\omega$ is the output function, $\omega$: $Q \times \sum \rightarrow \Gamma$ |
| $q_0 \in Q$ is the start state |

As opposed to FAs, Mealy machines are classified as transducers, where instead of recognizing a string and determining whether it is in *L(M),* instead output is generated by the machine depending on the input string and the output function $\omega$. For Mealy machines, the output function is done along transitions, such that $\delta$ and $\omega$ could be represented by a single hybrid function $T: Q \times \sum \rightarrow Q \times \Gamma$. The JFLAP interface represents the Mealy machine as such, identical to a FA, but with each transition taking two symbols (one input, one output) instead of one. Figure 2 shows an example Mealy Machine with two states and one transition (*a; b*)

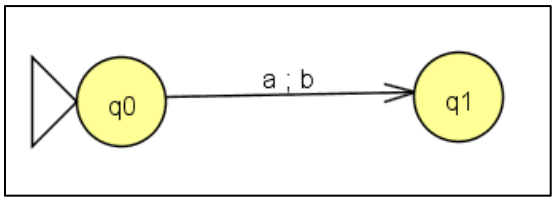


Figure 2: A basic Mealy Machine

## 2.3 Moore Machine

Similar to the Mealy machine, Moore machines are formally defined as a sextuple $M = (Q, \sum, \Gamma, \delta, \omega, q_0)$ where:

| |
|---|
| $Q$ is a finite set of states |
| $\sum$ is a finite input alphabet |
| $\Gamma$ is a finite output alphabet |
| $\delta$ is the transition function, $\delta: Q \times \sum \rightarrow Q$ |
| $\omega$ is the output function, $\omega: Q \rightarrow \Gamma$ |
| $q_0 \in Q$ is the start state |

Where Moore machines differ from Mealy machines is their output function. Instead of generating output based on transitions, each state is instead given an output. When that state is reached, the output is generated.

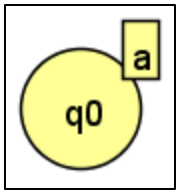


Figure 3: A Moore Machine state

## 2.4 Pushdown Automata

Similar to FAs, JFLAP contains the possibility for both deterministic and non-deterministic pushdown automata (PDA). Formally, $M = (Q, \sum, \Gamma, \delta, q_0, Z, F)$ where:

| |
|---|
| $Q$ is a finite set of states |
| $\sum$ is a finite input alphabet |
| $\Gamma$ is a finite stack alphabet |
| $\delta$ is the transition function, $\delta$: $Q \times \sum^* \times \Gamma^* \rightarrow$ a finite subset of $Q \times \Gamma^*$ |
| $q_0 \in Q$ is the start state |
| $Z \in \Gamma$ is the start stack symbol |
| $F \subseteq Q$ is the set of final states |

Different from the simple FAs, PDAs utilize a stack as extra memory to allow for a more powerful machine. The stack is initialized with a single symbol, the bottom-of-stack or start stack symbol, *Z*. Transitions contain push and pop symbols as well as a read symbol, which edit the stack while parsing a string. PDAs may accept strings in two ways, depending on preferences, either by reaching a final state after parsing the entire input, or by emptying the stack. The language *L(M)* of a PDA is less restricted than that of FAs, and is classified as the context-free set of languages that can be represented by context-free grammars.

## 2.5 Turing Machine

Formally, a TM $M = (Q, \Sigma, \Gamma, \delta, q_0, b, F)$, where:

| |
|---|
| $Q$ is a finite set of states |
| $\sum$ is a finite input alphabet |
| $\Gamma$ is a finite tape alphabet |
| $\delta$ is the transition function, $\delta$: $Q \times \Gamma^n \rightarrow Q \times \Gamma^n \times \{L, R, S\}^n$ |
| $q_0 \in Q$ is the start state |
| $b \in \Gamma$ is the blank symbol |
| $F \subseteq Q$ is the set of final states |

A Turing Machine (TM) is a more powerful automaton than the DFA or NPDA. It was originally theorized by Alan Turing to be able to compute any mathematical function for which there exists an effective algorithm, and uses storage in the form of read/write tapes, with each cell on the tape holding a single symbol. Turing machines can be constructed with multiple tapes, though they are equivalent in power to a single-tape machine. Each transition is specified with a read and write symbol, and uses an extra specification, which represents moving a tape's head one cell left, right, or staying where it is (L, R, S). The machines function as both transducer and acceptor, and comes in many other varieties besides single and multi-tape, though JFLAP does not delve into them (these include multi-track, semi-infinite tape, and multi-dimensional). JFLAP does, however, allow for the creation of *blocks*, which function as a single state in a TM, but in truth is an entirely separate TM, a *sub-machine* to be reused, add functionality, and allow the user to create more complex machines while maintaining levels of abstraction. Finally, *L(M)* of a TM *M* is in the even less restricted family of unrestricted languages, and JFLAP includes an algorithm to transform a TM into an unrestricted grammar.

## 2.6 Grammar

A Grammar is represented by the tuple $G = (V, T, S, P)$ where:

| |
|---|
| $V$ is a finite set of variable symbols |
| $T$ is a finite set of terminal symbols |
| $S \in V$ is the start symbol |
| $P$ is a finite set of production rules of the form: |
| $(V \cup T)^* \; V \; (V \cup T)^* \rightarrow (V \cup T)^*$ |
| $V, T \neq \emptyset$ and are disjoint |

Grammars are another way of representing a formal language. Different types of grammars, such as right-linear, regular, context-free, and unrestricted, are equivalent to different automata. A grammar accepts an input string *w* if there exists a derivation, a finite ordered number of production rules that, when applied to a string of symbols starting with the start symbol *S*, *w* is eventually generated. JFLAP v7.0 includes many parsing algorithms for determining this, depending on the family of grammar, including brute force parsing, CYK parsing, LL(1) parsing, SLR(1) parsing, and user generated parsing. It also contains algorithms for testing the type/family of grammars, transforming context-free and regular grammars into Chomsky Normal Form (CNF), and conversion from a regular grammar to an FA and from a context-free grammar to an NPDA.

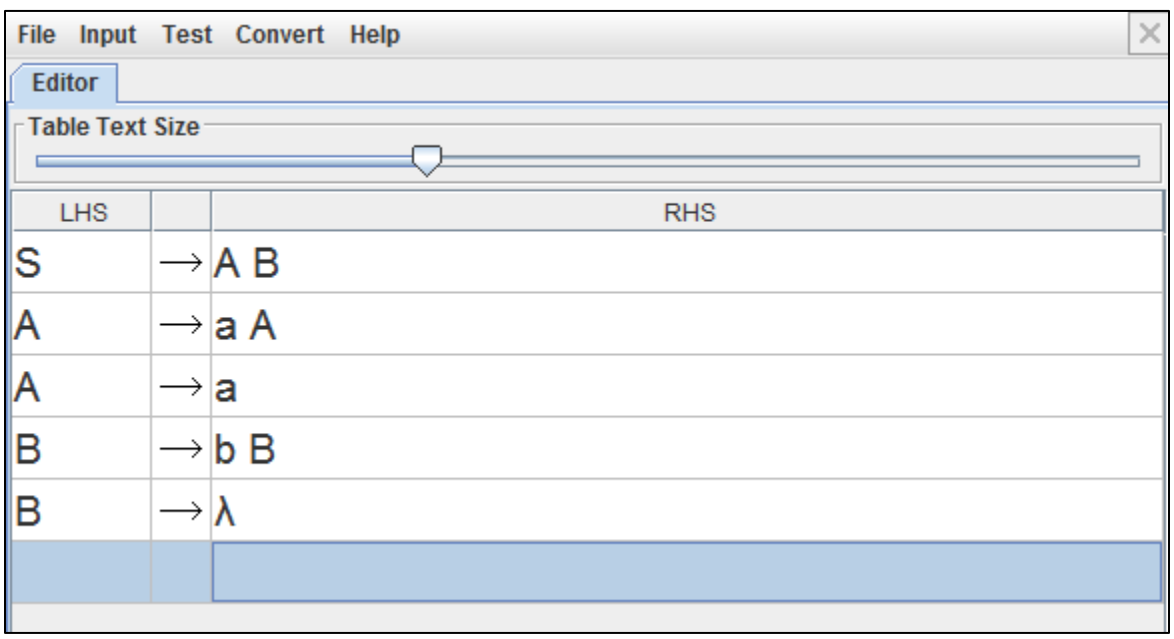


Figure 4: A sample grammar

## 2.7 Lindenmayer System

A Lindenmayer System (L-System) is a type of grammar in which an *axiom*, or starting string, is provided and production rules are applied in parallel, generating a longer string. Through the use of Turtle graphics, special symbols in the grammar allow for a graphical representation of the string at each step, displaying fractals, simulating plant growth, and creating a visual representation of a formal language. Formally, an L-System is defined as the tuple $(\sum, A, R)$, where:

| |
|---|
| $\sum$ is a finite alphabet of symbols |
| $A \in \sum^+$ is a finite axiom |
| $R$ is a finite set of production rules of the form: |
| $(a, b, c) \rightarrow d \mid a, c \in \sum^*, b \in \sum, d \subset \sum^*$ |

For any production $((a, b, c) \rightarrow d) \in R$, if $|d| > 1$, the L-System is a stochastic L-System, and if $a \neq \lambda$ or $c \neq \lambda$, it is a contextual L-System. JFLAP v7.0 allows for creating named parameters as well, such as angle, distance, and color, which will affect how the Turtle graphics render the axiom at a given step. Figure 5 shows the progression of an L-System with three renderings of strings, and figure 6 shows a more complex one.

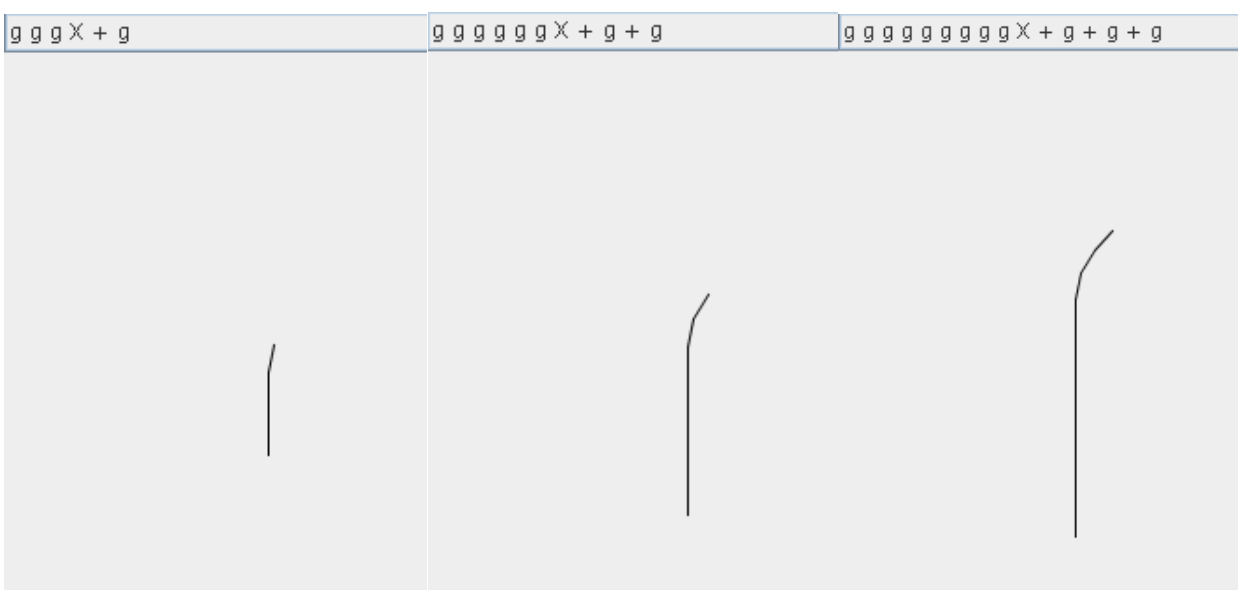


Figure 5: Sequential derivations of an L-System

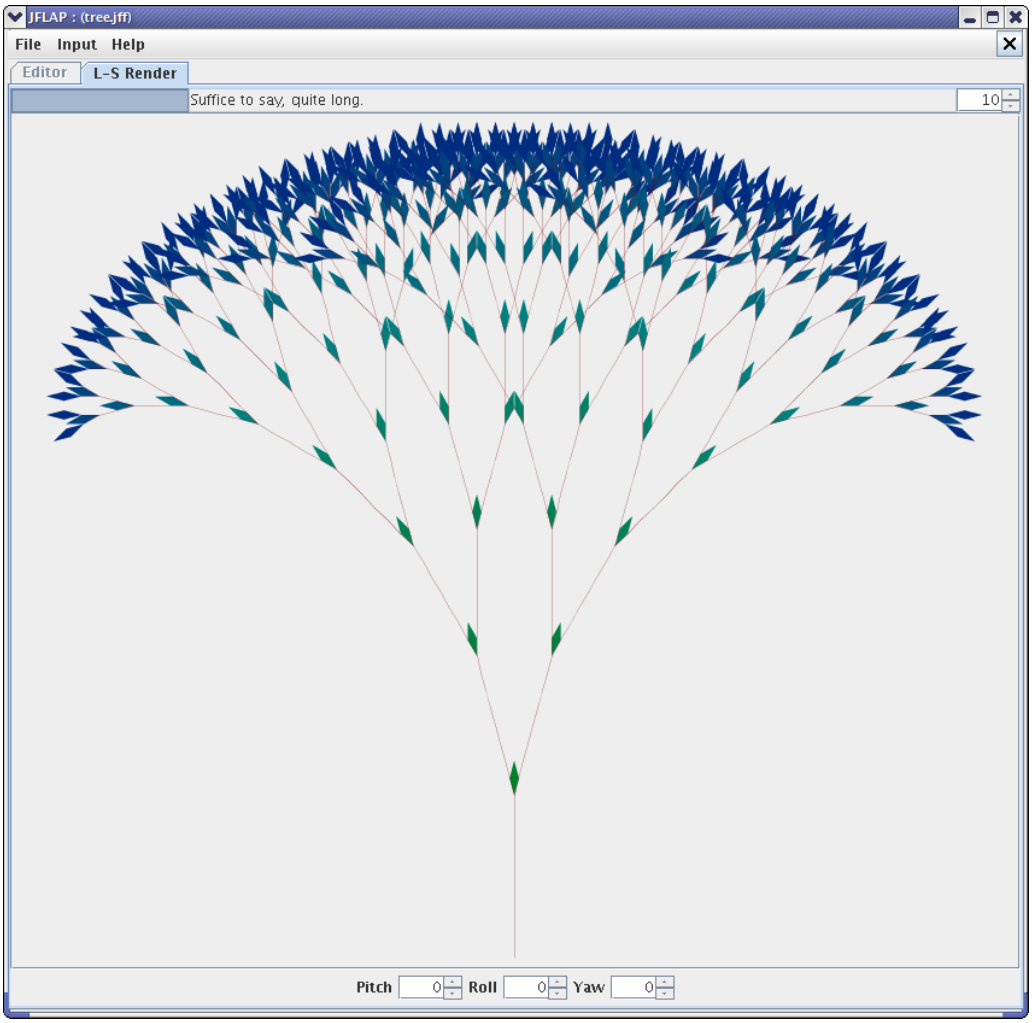


Figure 6: An L-System derivation rendering manipulating polygon color

## 2.8 Regular Expression

Another representation of regular languages along with regular grammars and FAs is the regular expression (RE). They provide a way of defining languages through the combination of symbols of its input alphabet $\sum$, grouping characters such as parenthesis, and concatenation (·), alternation (+), and Kleene star (*) operators. With these operations and $\sum$ along with three rules below [Lin 2006], a formal definition is attained.

1. *$\emptyset$, $\lambda$, and a $\in \sum$ are all regular expressions, called primitive regular expressions.*
2. *If* $r_1$ *and* $r_2$ *are regular expressions, so* are $r_1 + r_2$, $r_1 \cdot r_2$, $r_1^*$ *and* $(r_1)$.

*3. A String is a regular expression if and only if it can be derived from the primitive regular expressions by a finite number of applications of the rules mentioned in (2).*

JFLAP uses this definition to describe regular languages with REs, though instead of using a concatenation symbol, symbols directly next to one another (i.e. *ab)* are implicitly concatenated. Algorithms to convert REs to regular grammars and DFAs are included as well.

### 2.9 Pumping Lemmas

The final two modules included in JFLAP v7.0 are the Regular and Context-free Pumping Lemmas. Based on the approach from Linz's *An Introduction to Formal Languages and Automata*, JFLAP treats the pumping lemmas as a two-player game against the computer to demonstrate how the lemmas function. It is used on a variety of pre-provided languages to prove whether or not that language is regular or context-free. By allowing for the user to take either role in the game (trying to prove or disprove each), it helps the user interact more in-depth with the complex lemmas.

# 3. Limitations of JFLAP v7.0

While JFLAP is one of the most widely used tools for aiding in education of automata and formal language theory, as a program developed over the course of two decades, there are some limitations and possibilities for extension to improve upon its capabilities as a teaching tool.

The most prevalent limitation, as discussed in [Gen 2012], is the fact that all symbols within JFLAP v7.0 are restricted to single alpha-numeric characters, and all combinations of symbols are, by extension, Java Strings. The first issue that arises due to this fact is that there are a maximum number of variables and terminals (in the case of grammars) or symbols (for all other formal definitions) that the user can input. This rules out complex formal definitions that require symbols beyond that scope, along with more symbolic languages where, instead of a single character, multiple characters or even words might be used to represent a symbol (such as a real-world programming language or a description of a sentence), or changing the start variable to something besides the reserved *S*. What is less obvious is the fact that due to dealing with String

literals, and not a more flexible abstraction, some algorithms, notably conversion algorithms such as TM to Unrestricted Grammar, utilize direct String manipulation. This means that, hidden from the user, special characters are used as markers to split specific Strings on or provide functionality that is nowhere explained. To the user, this means that the program will not let them input characters with little to no explanation, or will simply cause errors when the algorithm runs and finds more special characters than it was expecting.

Another issue facing JFLAP v7.0 is the lack of explicit formal definitions. As explained in the previous sections, almost every module relates to a formally defined concept. In textbooks that don't utilize the visual representation capabilities of programs such as JFLAP, it is with these textual formal definitions that students learn the topics. JFLAP makes no attempt to relate its visualizations with these formal definitions, creating a separation between what sources may be available, and obscuring the components of what users are working with, which goes against the ideas of software visualization.

Similarly, there are components, particularly some of the parsing algorithms, which are available for the users to utilize, but not to learn. As a teaching tool, the educational value of any addition to the program is diminished if the user has no interaction with what is being run. The most notable of these is the CYK parsing algorithm, which in its current state demonstrates no part of the algorithm itself, instead only showing the results of the parsing on a given input string and grammar. Another is the Brute Force parsing algorithm, whose visualization tells the user how many nodes have been generated as it is run without further explanation or representation.

A more user friendly interface is also something that can be expanded upon, with undo and redo capabilities in all modules (currently it is only available for automata), magnification of textual components, easier navigation through menus, and user set preferences beyond the current version's current abilities.

Finally, a more unified codebase, with logical layout and documentation, is something the current version is extremely lacking in. As subsequent additions have been done over the years by different students with little overlap, much of the codebase lacks integration and has been

designed and coded with dissimilar styles as the years have gone on. As much of the modules share similar structure, as shown with their similar formal definitions, a more unified hierarchy would be beneficial in reducing code duplication and improving the ability for others to become familiar with both the code and platform as a whole.

# 4. Research Goals

The goals of this project can be divided into three sections that, though somewhat different in their aims, work together to create a more flexible version of JFLAP.

## 4.1 Restructuring JFLAP Hierarchy

The first goal of the research had to do with the completion of work set out in [Gen 2012] relating to the reorganization and rewriting of the JFLAP codebase to a more unified hierarchy including the *SymbolString* paradigms. By allowing for individual symbols to be represented as an abstraction of a Java String, the limiting factors of previous JFLAP versions will be dealt with, as will issues dealing with String manipulation in algorithms. Expanding on this to include explicit formal definitions, an easier interface to manipulate symbols, alphabets, and formal definitions, and a more integrated codebase will help to finalize what [Gen 2012] planned for the extension of JFLAP.

## 4.2 Implementation and Improvement of Graphic User Interface

While the restructuring is certainly an optimal goal, not enough has been done as far as creating a GUI that is familiar to previous users, more flexible and user friendly in its implementation, and integrated with the new design and code restructuring. This goal includes the implementation of GUIs for all automata, regular expressions, L-Systems, pumping lemmas, algorithms, and derivation trees as they are now represented in the new model. It also includes the extension of graphic tools to allow for improved preferences, control, and manipulation in a more logical layout to help both new and previous JFLAP users, while not adapting the previous versions' appearance to a degree that would discourage or confuse people used to the program.

## 4.3 New Functionality

The final goal set out in this research is extending the JFLAP program to include novel capabilities that will aid in JFLAP's capacity as an educational tool for automata and formal language theory. This includes the implementation of a visualization of the CYK parsing algorithm, a more user-relevant visualization of the Brute Force parsing algorithm, a new component that can generate strings in a language (for grammars, automata, and regular expressions), an overhaul of the Turing machine block system, and the capacity to load files created with previous versions as well as v8.0 (as the XML format for saved files in the new version will be fundamentally changed by the new code hierarchy).

# 5. Restructuring JFLAP Hierarchy

The creation of the JFLAP v8.0 codebase began in 2011 under Julian Genkins [Gen 2012] to improve upon JFLAP v7.0's flexibility. The first part was the transition from characters to *Symbols*. This brought about the transition from String literals (such as input strings in automata or derivation strings in grammars) to *SymbolStrings*, which are implemented as an ordered list of *Symbols.* The API allows for backwards compatibility, as single character *Symbols* are treated no differently from multi-character ones, while providing the capability of complex *Symbols*, which in a *SymbolString* are able to be interpreted as well. *Symbol* was also subclassed to include *Terminals* and *Variables* for grammars, so specific entries could be treated as one or the other.

An extension upon this use of *Symbols* provided the building blocks to the explicit formal definitions in the form of *Alphabets*. Before any *Symbol* can be used in a formal definition, it is first added to the appropriate *Alphabet* (such as a TM's *TapeAlphabet* or a grammar's *TerminalAlphabet*). These components, as well as others like *StartSymbol*, are able to be graphically represented at the base of each formal definition to be directly modified and examined, which reveals the inner workings of the program to aid in understanding, as well as provide a common interface throughout the entire package.

Similar work was done to integrate the rest of the codebase to this standard, including the implementation of an *Algorithm* hierarchy designed to provide abstraction and a common interface to all algorithms within the package. More importantly, a much stronger MVC architecture has been implemented to separate model code, such as a DFA, from graphical representations, such as the automata's graph. This is possible due to the use of explicit formal definitions, as the model now utilizes the actual definitions one would find in a textbook. While this does complicate some of the code and require extra information to be stored on saving, it allows for direct access and manipulation of objects within the model without using the JFLAP GUI. This provides the possibility for users who may not be interested in the graphic interface to work with or extend only the model, such as directly parsing multiple strings in a grammar, while also providing a clean way for future JFLAP researchers to implement new modules within the program.

# 6. Implementation and Improvement of GUI

After the completion of the new model code, one of the key goals of this research was to create a new version of the JFLAP GUI that made significant improvements upon the previous version's standards while not deviating from them too much. As JFLAP is constantly under development, it is important to keep the layout of it recognizable and familiar to users, especially instructors, while providing new functionality and improved usability.

## 6.1 Automata GUI

One of the most significant portions of the JFLAP program is its wide variety of automata and the algorithms it can run on them. More importantly, while other formal definitions, such as regular expressions or grammars, can be represented by textual descriptions, it is advantageous to represent automata by a much more graphic means. Along with this graphic representation arises the need for the user to be provided with proper tools to create, edit, and experiment with an automaton, and for these tools to function in an intuitive manner. In the creation of the new automata GUI, one of the biggest concerns was this idea of improving on the logical workings of the editing tools.

The first step, after the initial creation of the editing panel, was a modification to the State Creation tool, which in previous versions can create a new State at the moused-over point when clicked. This modification was to allow the right mouse click to actually have a function, which is moving States that have already been created (which was previously only doable with the arrow tool). Following a similar logic, that of expanding in a rational nature, as our new version of JFLAP differentiates between Turing machines that utilize States and those that use Blocks, the shape of the Blocks was changed to a square instead of a circle.

The way transitions are maintained and edited was improved upon as well. In previous versions, should multiple transitions go from one state to another, only one of them would actually be linked with the edge in the graph. This meant that if the user wanted to drag the edge to curve differently, only one of the text boxes would move and the rest would stay where they were initially. This was obviously not intended, as the edge represents all of the transitions, and the issue was fixed, with all textual descriptions of the transitions moving when an edge is changed. Also, the way an edge is edited was modified slightly. Previously, one had to click along the arrow, which would highlight it and reveal its “control point” which could then be manipulated. Instead, our new version allows for clicking and dragging anywhere along the edge to change its shape. Figure 7 shows the difference.

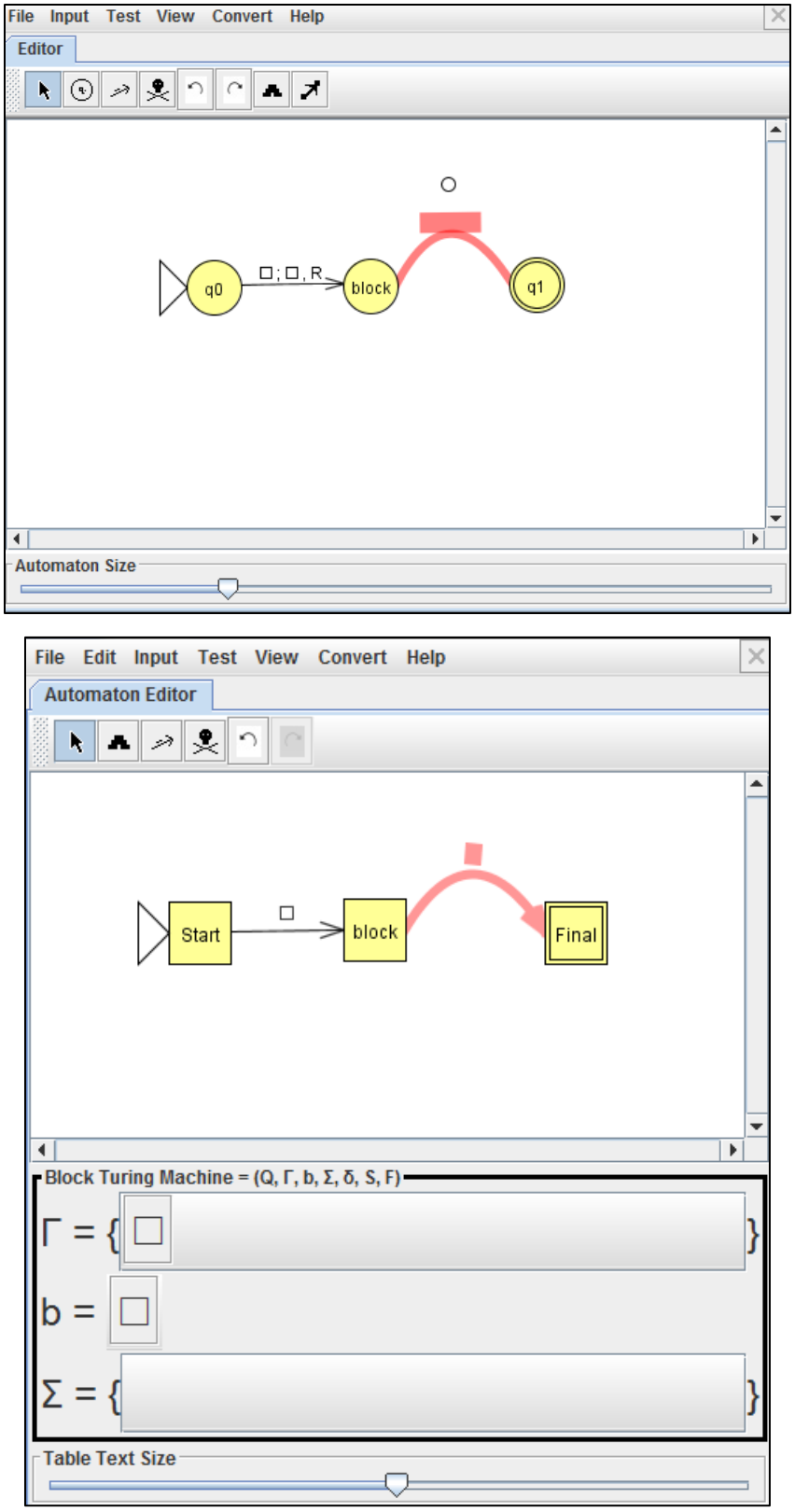


Figure 7: Top pictures JFLAP v7.0 manipulation of a transition in a TM utilizing a block. Bottom shows v8.0 manipulation of a similar Block TM. Note the control point in the top picture.

To increase usability, the capacity for users to select multiple objects (such as states, edges, transitions, and notes) within the Automata Editing Panel was expanded. In v7.0, one can click a space with nothing in it and drag to create a selection rectangle, which was preserved in the new version. In addition, the ability to select (and deselect) any object within the automata panel was added. Users now can simply hold down the shift or control key on their keyboard and left click with the arrow tool to select multiple objects. This allows for users to select exactly the objects they want to manipulate without having to rely on getting a rectangle perfectly around them. In addition to this, with the select feature active, a much easier to use delete function was added, so instead of having to manually click on each object with the delete tool active (though it is still a possibility) a user may now simply press the delete key on their keyboard with whatever they want selected (this also allows for multiple deletes at a time).

JFLAP v7.0 had a fairly simple interface when it comes to manipulating the placement of states, transitions, and notes in an automaton. Due to this simplicity, and the inner workings of Swing (the GUI package native to Java), users were able to drag objects off screen (either to the top or left of the canvas area) and be unable to access them again. Version 8.0 instead adjusts existing objects in a consistent manner, not allowing for any text or visual objects to be placed somewhere the user would be unable to access them.

Finally, the capabilities provided through the basic undo/redo implementation in v7.0 were extended to include all possible edits made using the new interface, including creating, editing, moving, and deleting of transitions, notes, and states, as well as applying layout algorithms to the graph.

## 6.2 Regular Expression GUI

The regular expression interface is one of the simplest modules in JFLAP v7.0, and with only a few changes, it remains the same in v8.0. The significant additions are the capability to undo/redo changes in the expression, verifying changes within the expression so that an incorrectly formatted input isn’t accepted, and the alphabet bar common to the rest of JFLAP v8.0’s formal definitions. You can see the comparison in figure 8.

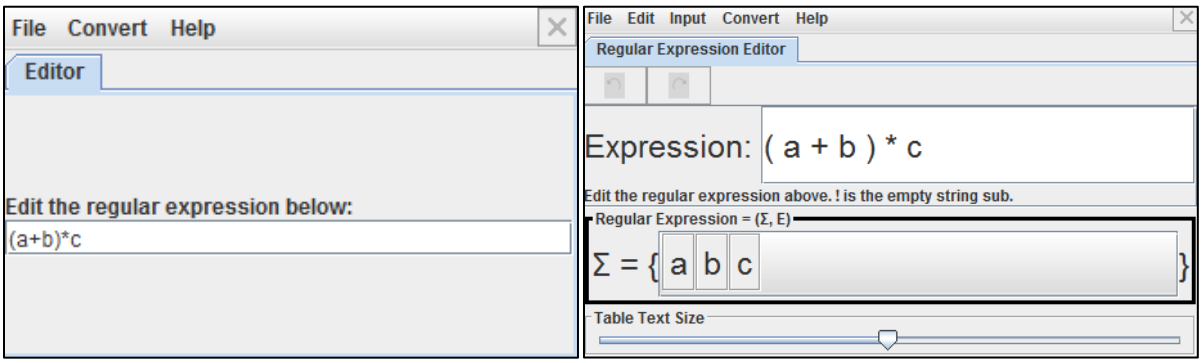


Figure 8: Regular Expression editing panel in JFLAP v7.0 (left) and v8.0 (right)

## 6.3 Lindenmayer System GUI

While L-Systems are somewhat disconnected from most of the JFLAP classes, the increased capacity of symbols definitely plays a role in the updated version. In fact, the way that L-Systems previously interpreted symbols prior to this update was somewhat similar to what we have now for the rest of JFLAP, though more restricted and lacking its complexity. Each symbol, from an L-System perspective, is simply a set of one or more characters that, when placed in a "sentence" with other symbols (each separated with spaces from one another), form axioms and production rules similar to that of a normal grammar. There are a large number of symbols that hold special meaning in L-Systems for JFLAP, and have to do with the rendering of these symbols to produce the graphic output (sample symbols include g, f, #, %, etc.). While L-Systems were fairly separated from the rest of JFLAP, the capabilities of having complex variables and terminals brings them much closer.

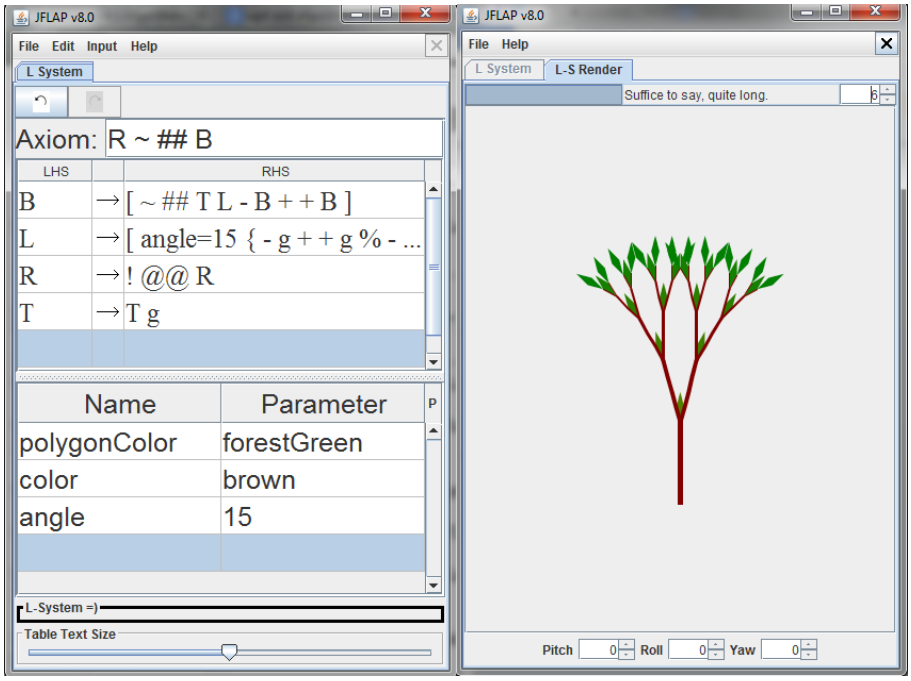


Figure 9: An L-System and the resulting rendering

JFLAP v8.0 has a fully supported L-System setup that incorporates the hierarchy of the rest of the package. One detail worked out was the fact that there was a "secret menu" that could be located by clicking the small grey area above the scroll bar of the parameter table to quickly insert values that hold specific meaning (such as *distance*, *color*, or *angle*) into the parameter table. While nothing short of redesigning the parameter table would allow for a full menu to be visible, a quick change from the area being a JFrame to being a JLabel allowed for a single letter (P for parameter) to be displayed in the area, so users might think to click on it, which can be seen on the left half of Figure 9.

There remains to be work still needed on L-Systems, including making it a more concrete formal definition as our code hierarchy dictates. L-Systems could further be improved upon by increasing its capabilities to mirror some of the topics discussed in *The Algorithmic Beauty of Plants* [PrL 1990], such as Stochastic L-Systems, where a probability is given to a specific production rule (where productions with the same left hand side have probabilities that sum to 1), the addition of more complex shapes (such as circles to simulate buds of a plant), and a better formulated and easier to understand format for context-sensitive L-Systems. In their current form, any context sensitivity is to be noted in conjunction with numbers specifying the symbol to

be replaced. If we want the rule that would appear in a grammar as $a\ b \rightarrow\ a\ d$, we must denote this as two separate productions, $0\ a\ b \rightarrow a$ and $1\ a\ b \rightarrow d$. This is not the best design, as it is counterintuitive, limits the use of numbers as symbols, and is not well rationalized besides from how it was originally coded. We were unable to deal with a complete redesign in this time frame, but we did design a renderer for cells on the left hand side of the production table for L-Systems such that when a context-sensitive rule is entered, it displays it with a bolding of the letter that is to be replaced while hiding the extraneous number (i.e. $0\ a\ b$ on the left hand side would be displayed as $\boldsymbol{a}\ b$, and $1\ a\ b$ would be $a\ \boldsymbol{b}$). We see in Figure 10 an L-System that will propagate the b to the right until it reaches the end.

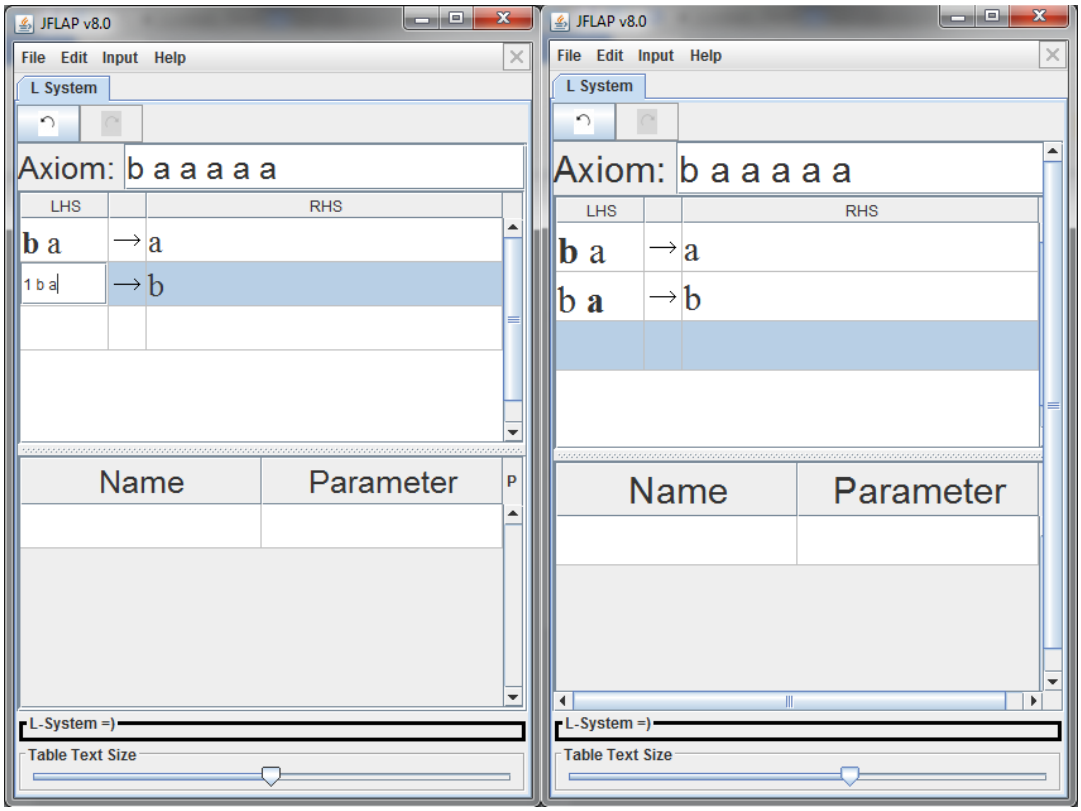


Figure 10: A simple context-sensitive L-System being edited by the user (left) and with cells rendered (right)

Finally, an adjustment to the way JFLAP saves images of L-Systems provides the capability of a fully rendered L-System to be saved in its entirety as a JPG, PNG, GIF, or BMP, as opposed to just the portion of it that the user window can visualize.

## 6.4 Pumping Lemma GUI

In their current form, Pumping Lemmas are completely hard coded, with specific languages already specified as the only options, and separated from everything else in JFLAP. The only real connection to the rest of the package is the GUI, which had minor issues with rendering in v8.0 with the new treatment of preferences and Symbols, and is now resolved.

We think that Pumping Lemmas could be modified similarly to what was done with the library Building Blocks for Turing Machines (which will be discussed in section 7). In the future, Pumping Lemmas can have work done to improve their flexibility and let users design their own. While the code itself could use some rewriting before v8.0 is released, Pumping Lemma improvements are currently at a low priority.

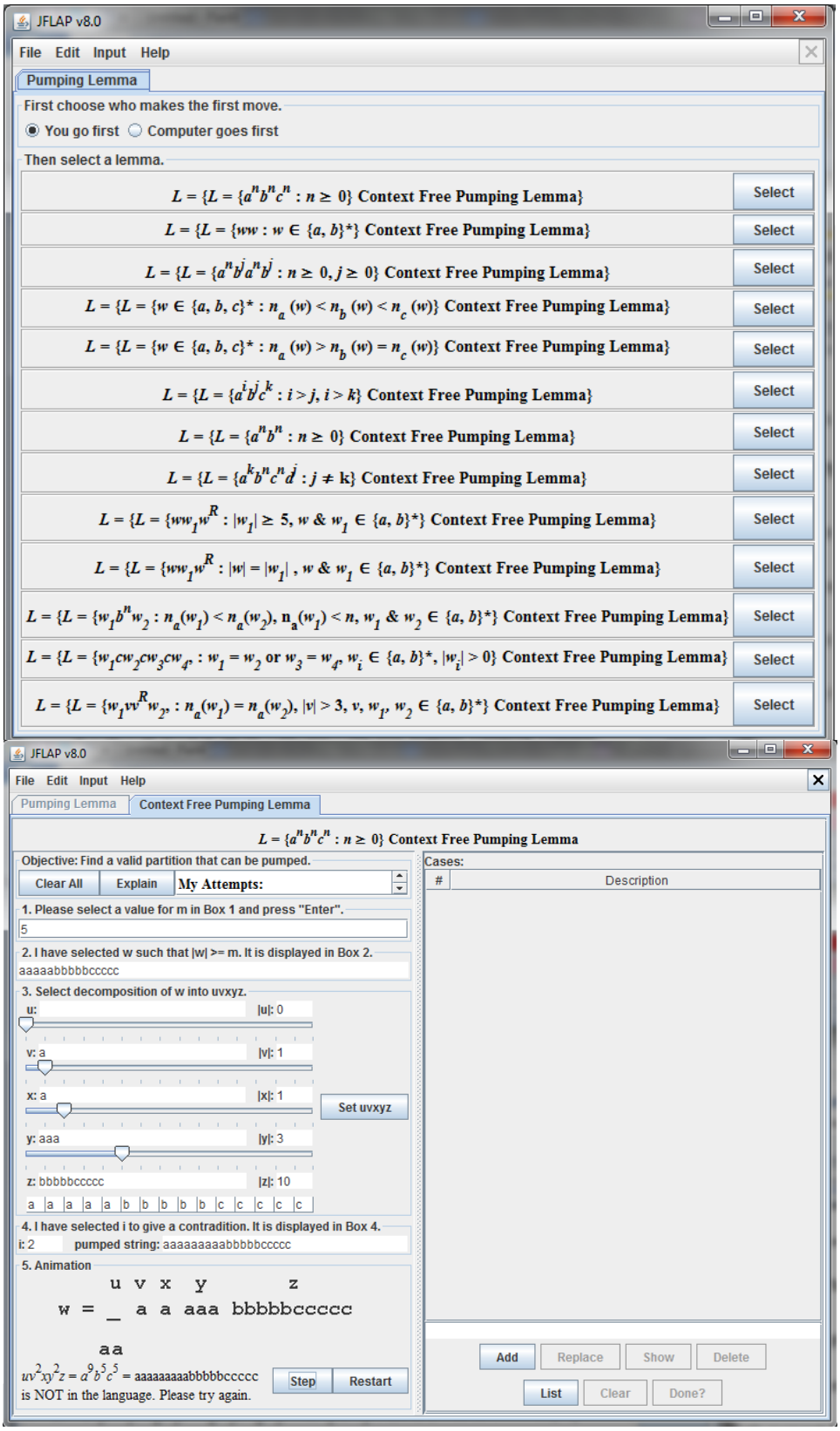


Figure 11: Context Free Pumping Lemma's menu and an example of a human first game

## 6.5 Algorithm GUIs

In addition to the various formal definitions, their graphic representations, and tools to manipulate them, one of the most important uses of JFLAP is the various algorithms that can be run to convert, test, and parse input on those definitions. While not yet completely finalized, the interfaces for all algorithms having to do with automata besides *Minimize DFA*, all conversion algorithms (automata to grammar, grammar to automata, FA to RE, and RE to FA), and two parsing algorithms (Brute Force parsing and CYK parsing) have been implemented. The Transform Grammar algorithm, LL(1) Parsing, SLR(1) Parsing, and User Parsing still need graphic implementation, but the foundation for all of them has been created.

A special panel known as the Derivation or Parse Tree has also been implemented, though significant changes would be encouraged in the future to improve upon the visualization algorithm. This panel is displayed after parsing is complete, as well as within some algorithms, and shows the derivation of a string in the form of a tree, as seen in figure 12.

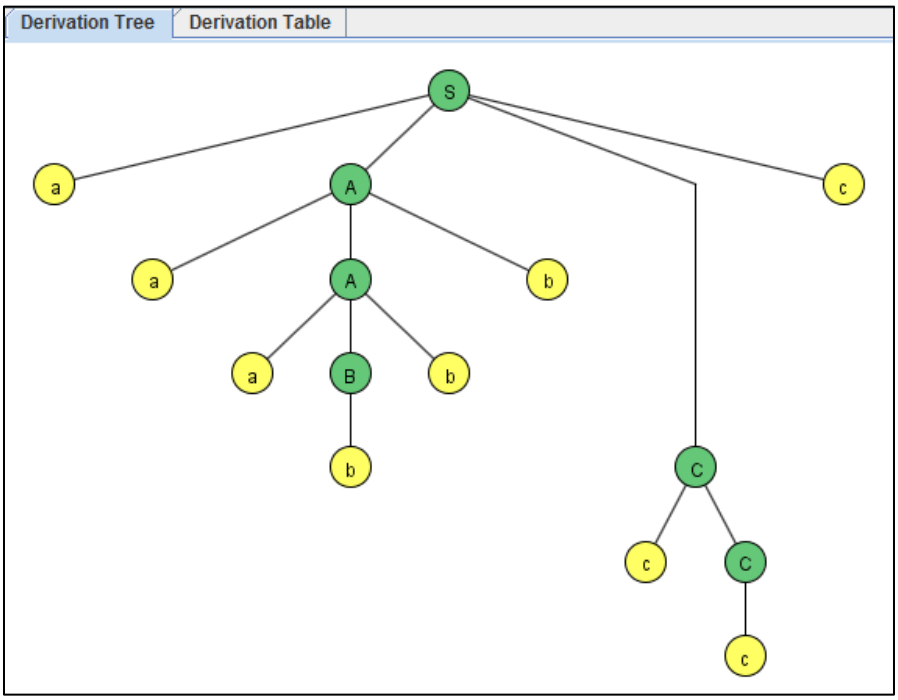


Figure 12: Derivation Tree of string *aaabbbccc* in an $a^n b^n c^n$ grammar.

## 6.6 Extension of Graphic Tools

One of JFLAP v7.0's shortcomings is the lack of flexibility when it comes to graphic representations. As much of this has to do with how Swing functions, and not with JFLAP code necessarily, not much was done in the past. However, to make the interface more user-friendly and similar to other programs users may be used to, some extensions have been implemented.

The first is an enhanced preferences menu. This menu allows for users to change a wide variety of options having to do with default values, such as L-System parameters, λ vs ε for the empty symbol, the direction of the CYK parse table, and the default mode (single-character, multi-character, or custom defined) for symbol recognition. Also, the ability to set the colors of various objects, such as default or highlighted states in an automaton, has been added, which we hope will provide users the ability to have JFLAP look the way they prefer, match textbooks, and perhaps help those with colorblindness.

The second extension is the improved dialog navigation and control. In the new version, all menus, popup dialogs, and save windows can be navigated in a variety of ways, as opposed to just with a mouse pressing the buttons. Users can use tab and shift-tab to cycle through options, the arrow keys to go in specific directions, the space and enter keys to press a button, and key bindings such as {y, n, c} (for a "yes", "no", "cancel" dialog) to select options throughout the program. While not notably obvious, we hope this provides users with a more complex, user-friendly, faster way to control JFLAP.

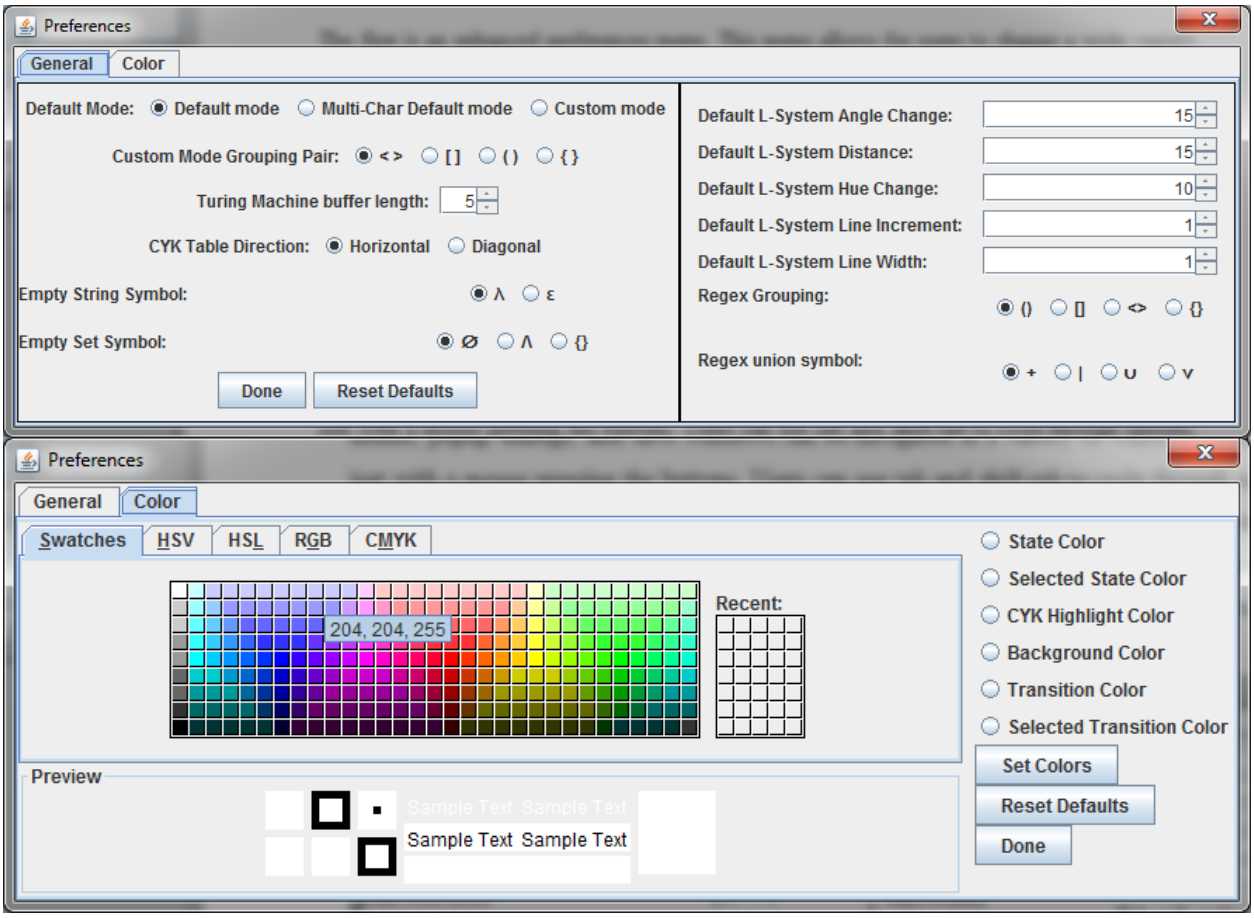


Figure 13: Preference panel in JFLAP v8.0

# 7. New Functionality

JFLAP is an ongoing project, and while the new version does need to have (or enhance upon) the capabilities of previous versions, it is the new functionality that makes it stand out as a more effective teaching tool in automata and formal language theory. These larger projects give new capacity to JFLAP v8.0 that will hopefully add to its use worldwide.

## 7.1 CYK Algorithm

The CYK (Cocke-Younger-Kasami) algorithm [CoS 1970, Kas 1965, You 1967] is a dynamic programming, bottom-up parsing algorithm for context-free grammars, in Chomsky Normal Form. It has a worst case running time of $O(n^3 |G|)$ (n is the length of the string being parsed and |G| is the size of the grammar). Essentially, the algorithm considers all possible subsequences,

starting with single symbols, then strings of 2 consecutive symbols, and so on, and checks every partition of that subsequence. As productions in CNF have the form $X \rightarrow AB$ (two variables on the right) or $X \rightarrow a$ (a single terminal on the right), these partitions are formed by splitting the subsequence into two sub-subsequences at each symbol and seeing if they can be parsed. When it is complete, the final string will be verified or rejected.

As defined in *An Introduction to Formal Languages and Automata* [Lin 2006], the algorithm is as follows:

Assume we have a grammar $G = (V, T, S, P)$ in CNF form and a string $w = a_1 a_2 \ldots a_n$. We define substrings as $w_{ij} = a_i \ldots a_j$ and subsets of $V$, $V_{ij} = \{A \in V : A \overset{*}{\Rightarrow} w_{ij}\}$.

To calculate $V_{ij}$ we note that $A \in V_{ii}$ if and only if there is a production $A \rightarrow a_i$. We can then see that for any $V_{ij}$, $A$ is a member only if there is a production $A \rightarrow BC$ where $B \rightarrow w_{ik}$ , $C \rightarrow w_{k+1,j}$ for some $k$ with $i \leq k < j$.

Then, to compute the table, we follow the steps:

1. Compute $V_{11}, V_{22}, \ldots V_{nn}$
2. Compute $V_{12}, V_{23}, \ldots V_{n-1\ n}$
3. Compute $V_{13}, V_{24}, \ldots V_{n-2\ n}$,

   etc.

$w \in L(G)$ if and only if $S \in V_{1n}$ .

The algorithm was included in JFLAP 7.0 as a way to quickly verify if a string was part of a language, but no visualization was included. The CYK algorithm would run in the background and (if the input string was verified) would generate a parse tree, but there was no indication of what was happening or how it got to that conclusion. After the refactoring of the model's code to fit our new framework and standards, our prime goal was designing a visualization that would help teach what the algorithm is doing at each step, and allow for user interaction, similar to our other parsing algorithms. While incorporating some of the tools JFLAP already had at hand, much of the CYK algorithm's representation (which is also flexible in how the user wants the

data displayed) is completely novel. Figure 14 shows a simple grammar with a few rules and the input string *aaab* being parsed using the CYK visualization.

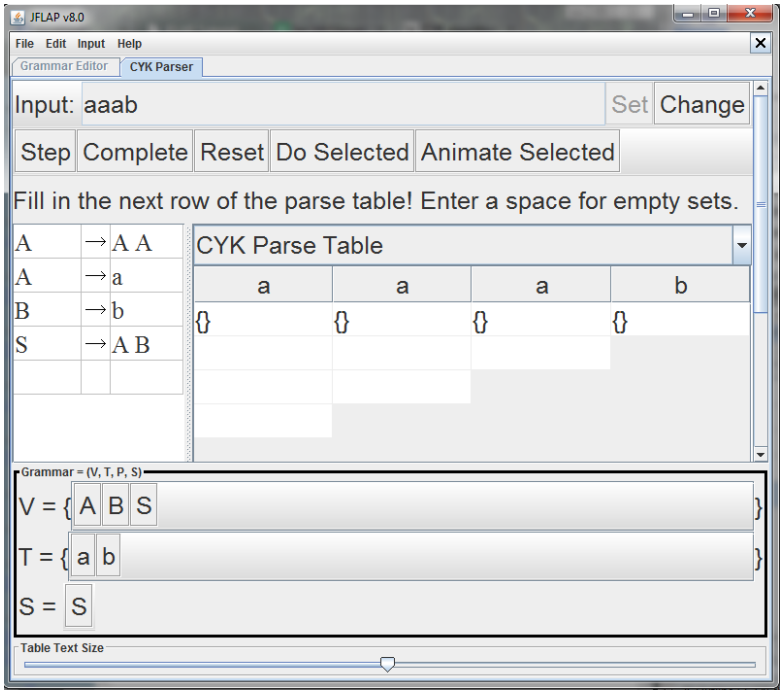


Figure 14: This is only one of the layouts, the other has the same functions, but looks and is filled slightly differently.

The parsed string is input in the top textbox, and once set generates the CYK table. The user can change this string at any time, but this will reset the table. As with the rest of JFLAP 8.0, there is a magnification slider that will allow for all text to resize. There is the line of text above the table that gives a brief explanation for what the user should do, and the dropdown menu will become active (with the finalized parse tree) once the table is filled and the string verified. The active row is shown by the empty brackets, and editing is only possible on active cells. The user can click on one of the active boxes and input the set of symbols that can create the given terminal (which will be highlighted). If the entered variables are completely correct, the box will remain the white color, while incorrect entries will be red. The use of color to encode the state of the data and to highlight activity by displaying incorrect boxes in red and the column headers of selected cells in yellow provides an improved aid for the user's understanding [BrH 1998]. By conveying these changes through the use of subtle but noticeable colors, we aim to enhance communication to the user without overwhelming them. The user also has the capability to (on a selected active cell) press the "Do Selected" button, which will automatically fill the cell, "Step,"

which will complete all cells on the active row, “Reset,” which will clear the table, “Complete,” which will complete the entire table, or “Animate Selected,” which we will talk about later. The table also has a built in renderer to display the entered set using set notation (so putting “AB”, “A,B”, “A B”, etc. in a cell would render as “{A, B}” and empty sets render as the empty set symbol Ø).

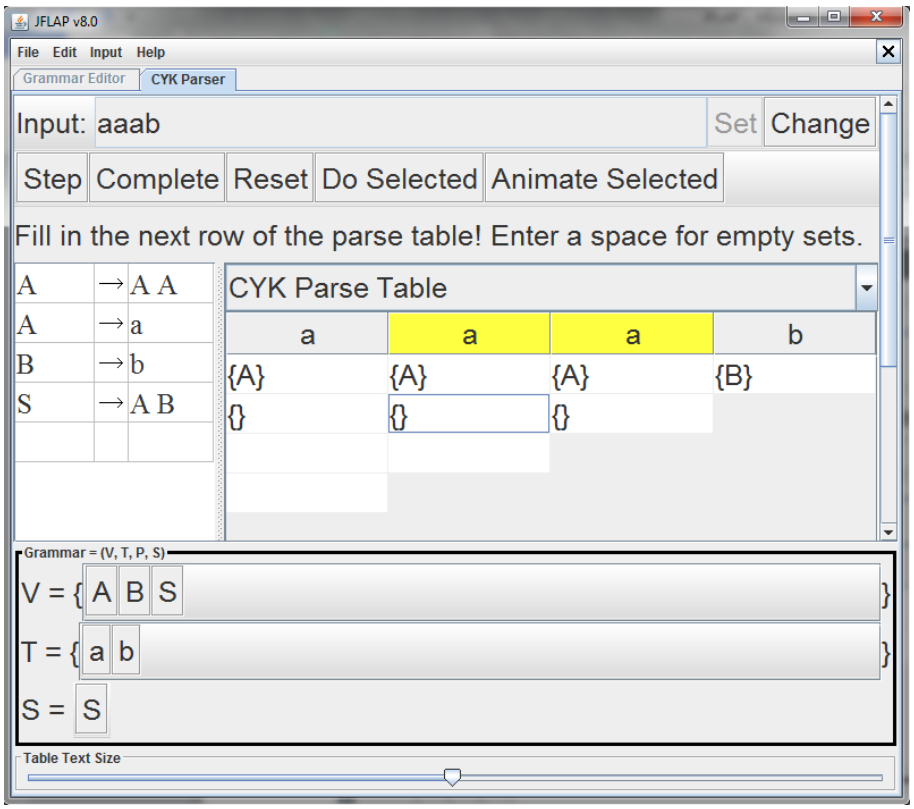


Figure 15: The table after completing the first row and selecting a box on the 2nd row.

When a row is completed, by either the software or the user, the next row becomes active and is filled with empty brackets. We see in Figure 15 that when a cell is selected (in this case, the second cell of the second row), the string that it represents becomes highlighted in the column headers. Each box represents the substring starting at its column with a length equal to the row + 1, so the selected box (1, 1) is the substring at index 1 of length 2 (in this case *aa*). For strings of length ≥ 3, each possible division must be considered, and this is precisely what the animation button shows to aid the user in choosing what symbols to consider.

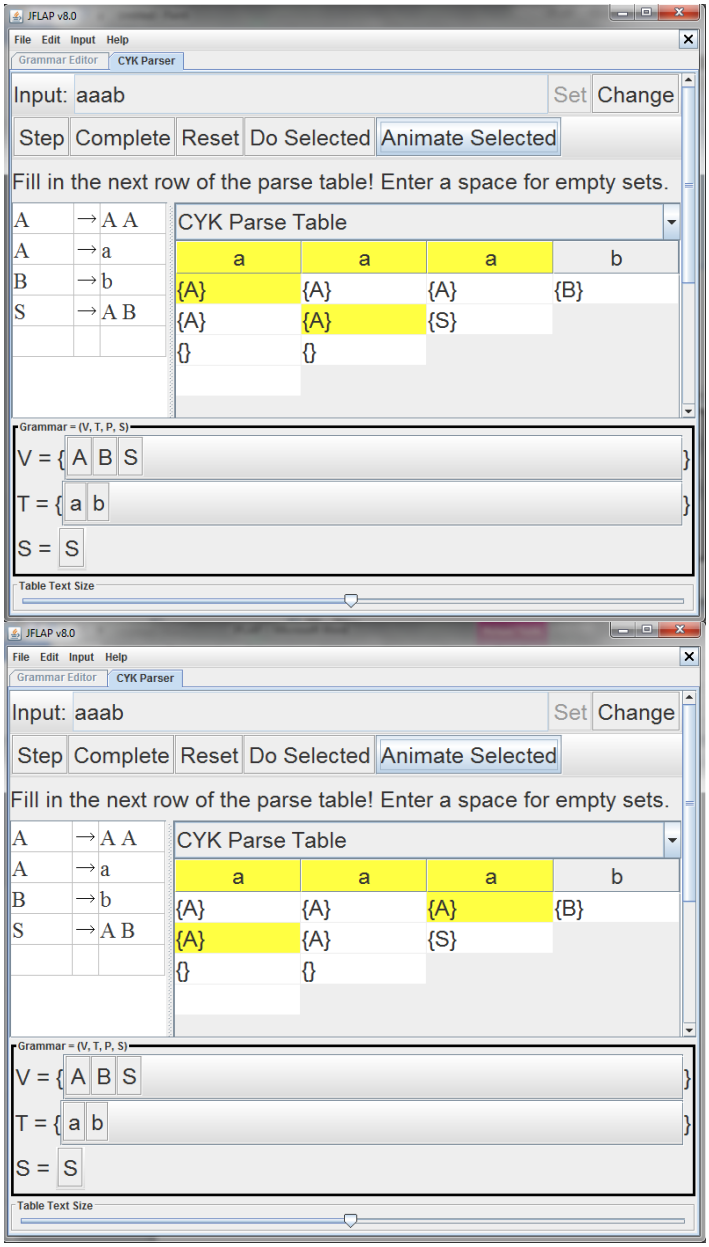


Figure 16: Two views of the "Animate Selected" feature of the CYK visualization.

In figure 16, the top view shows the table with box (3,0) (or the first box in the 3rd row with entry {}) selected and is demonstrating to look at productions with a right hand side containing a symbol from the set at (0, 0) as the first and one from (1, 1) as the second (in this case representing a production that stands for X→ *a aa*), and the bottom shows (1, 0) paired with (0, 2) (X→*aa a*), which we can see is simply A→A A in both cases. All pairings must be considered in the CYK algorithm, and this "Animate Selected" feature shows the user which pairs. This is simple to see in this case, but can be more helpful with more complicated strings and grammars.

The color used is quite neutral, and matches the color used to highlight the column headers. Each cell pairing in the animation stays highlighted for approximately one second to allow for enough time to notice the pairing without lagging. The goal was to produce a smooth animation that allowed gradual, continuous updates in the animation to provide for easier perception and understanding of the algorithm without taking too long and causing the user to become impatient [Sta 1998].

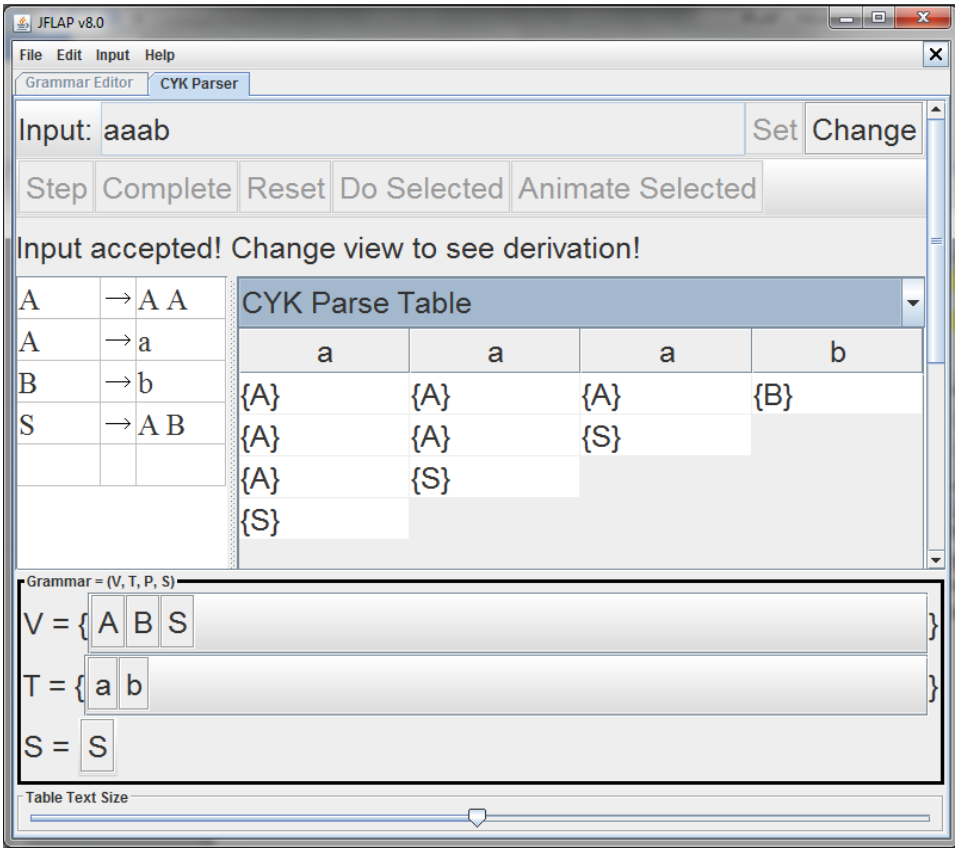


Figure 17: Completed CYK Parse Table

Figure 17 shows the completed table. Once completed, the buttons will become inactive. If the start variable is in the final box, then there is a possible derivation for the string, and the derivation tree becomes available from the dropdown menu, which functions as it does in the old versions of JFLAP. We can see both the derivation tree and derivation table in Figure 18.

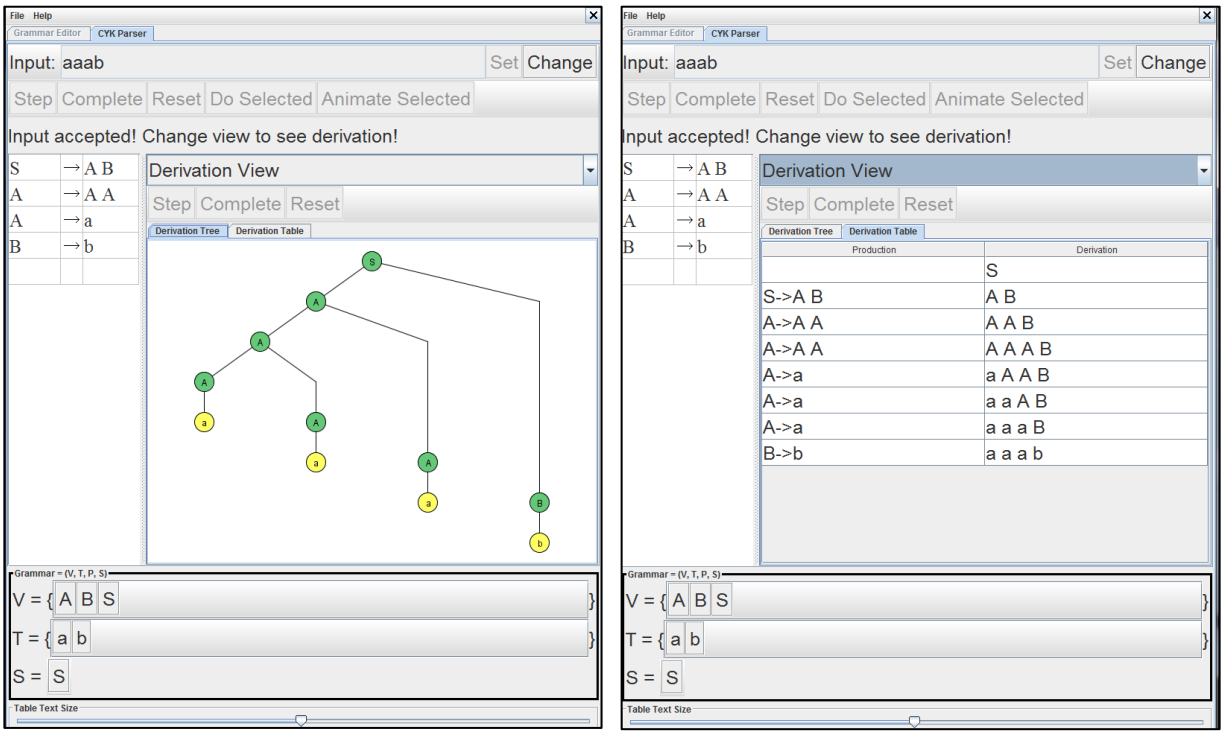


Figure 18: The derivation tree panel (left) and derivation table panel (right) of the CYK visualization.

There is also the capability of changing the layout such that instead of working by row, from top to bottom, the table is filled on the diagonal, from the middle to the top right corner. This capability is to provide for different textbooks and descriptions of the algorithm, it doesn't change the functionality but allows for flexibility depending on how the user wants to display the data (which can be changed in preferences). Our visualization provides an alternate visual and interactive view of the algorithm that may help students in understanding the algorithm. Future work could expand the visualization to work in other table layouts.

## 7.2 Brute Force Parsing and Language Generation

Another goal was focused on improving the Brute Force Parsing algorithm, and more specifically, its accompanying GUI. The algorithm is the least efficient parsing algorithm for grammars, but allows for computation on even the most complex unrestricted grammars that none of the other parsers can handle. Essentially, at each step, the algorithm contains a list of current derivations, starting with a single derivation containing the start variable. For each derivation, all productions are examined and applied individually (if the symbols on the left hand

side of the production are contained in the derivation) at all possible points in the derivation string to create new derivations. If at any point the target string is found, or the list of derivations becomes empty (i.e. there are no valid productions to apply to any strings) the algorithm terminates.

Along with a major cleanup of the code from JFLAP v7.0, the design on the Brute Force Parser made it fit into the Parsing hierarchy much more than before. While LL(1), SLR(1), and CYK all have some sort of table needed to fill out, the Brute Force Parsing algorithm before the code change simply ran, either until it found a working solution, rejected the string, or hit a limit of steps (as it can take a long time to run using brute force methods), and the only visualization consisted of a "node" counter that fluctuates. With our design of parsers in general, each one can now be done one "step" at a time, in this case one level of replacement in the brute force method (though by pressing complete, it will run somewhat similarly to the previous version). At each step in the algorithm, the user can see how many possible derivations there currently are and what they may look like, as shown in Figure 19 below. After a string has been accepted, the derivation tree pops up in the dropdown menu and the user may look through the various steps as in previous JFLAP versions.

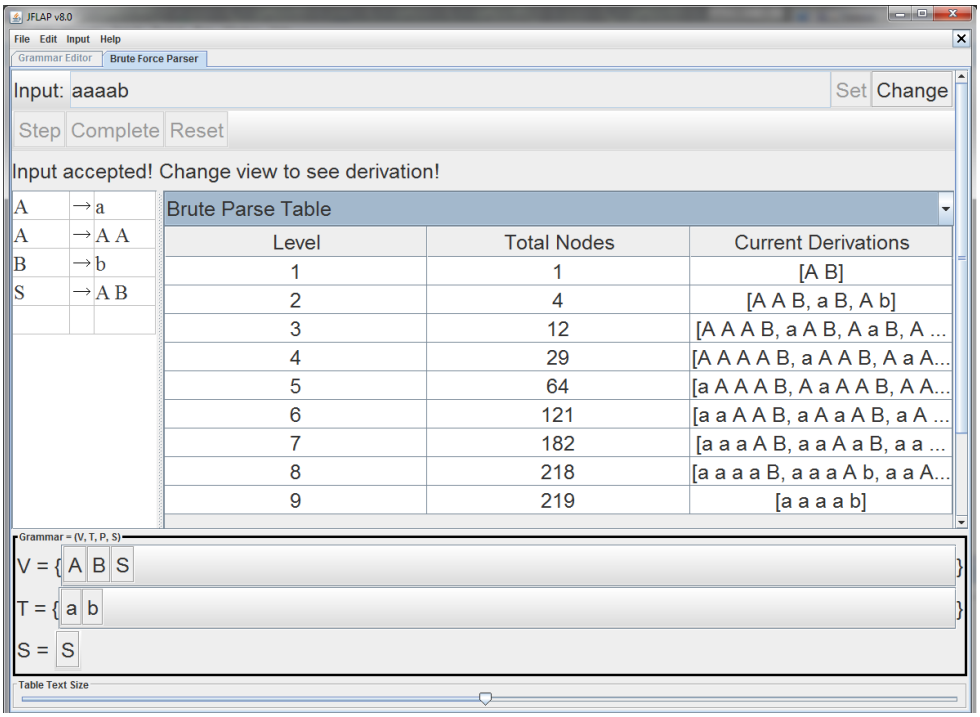

Figure 19: An example of the Brute Force paring table.

Having written these parsing algorithms, another addition to JFLAP was imagined, that of language generation. By passing in a grammar, or something that is equivalent to a grammar, the Language Generator will create either a specified number of strings that the language can produce, or all the strings of a specific length. The algorithm itself is not the most efficient, but it takes the ideas behind the CYK (for context-free grammars of infinite size) and brute force parsing (for any finite, context-sensitive, or unrestricted grammars) to produce meaningful results as to what a grammar can produce. We believe that this is useful for many grammars, as it allows the user to experiment with the grammar to see what can be generated. This change makes this algorithm interactive for the user at an introductory stage to formal languages, letting the user experiment with what one can do with languages.

Along with the algorithm, we came up with a very basic user interface that can be seen generating an $a^n b^n$ grammar in Figure 20 below. Here, the user has entered 6 as the number of strings to generate in the language, so the first six strings, *ab, aabb, aaabbb, aaaabbbb, aaaaabbbbb,* and *aaaaaabbbbbb* are generated. The user could have also chosen strings of *length* 6, which would have only returned *aaabbb.* If there are more strings than can fit in the view, a vertical scroll bar will appear, and if the strings are longer than the view, a horizontal one will appear as well. While there is no “maximum” number of strings one can generate, it is not a very efficient algorithm, so warnings will be displayed similar to the Brute Force Parsing when trying to generate strings that will take exponential time to generate.

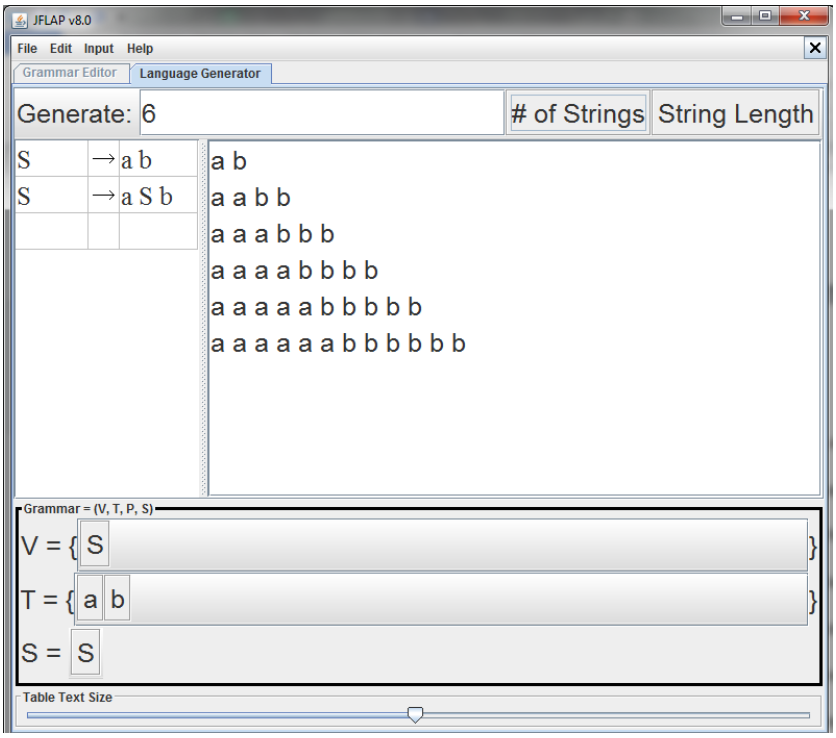


Figure 20: Language Generator running on $a^n b^n$ grammar.

## 7.3 Turing Machines

Along with these new visualizations and algorithms, we also worked on Turing machines, in particular an overhaul of the Building Block Turing machines to allow for greater flexibility, Universal Turing machine simulation, a conversion algorithm to transform a TM with Stay-options to one without, and a conversion from Turing machine to Unrestricted Grammar.

Block Turing machines are simply Turing machines with a level of abstraction, such that instead of states, they use pre-constructed Turing machines to do their calculation. For our design of Turing machines we wanted users to be able to construct them in a combination of ways: building them from scratch, using previously saved user-built machines imported as Blocks, or by using some of the library of Building Blocks that we designed. The previous version of JFLAP had something similar to this library, but they were pre-saved Turing machines identical to the ones a user could have constructed (and were therefore quite static in what they could accomplish, needing a completely new Turing machine for any differing functionality or alphabets). To increase flexibility, we instead coded a library of Blocks that take input (usually a direction and a symbol as specified by the user) at runtime to create specific blocks as the user

needs as opposed to having to load a very specific block from a file. These blocks include Copy, Start, Halt, Move (once either left or right), Move in a direction (R or L) until a specific Symbol is read, Move in a direction until a specific Symbol is NOT read, Shift tape (right or left), and Write a specific Symbol. The library blocks also carry the capacity of automatically updating as the alphabet of the TM using them is modified, ensuring that their functionality is retained. We can see these options as they are visualized in figure 21.

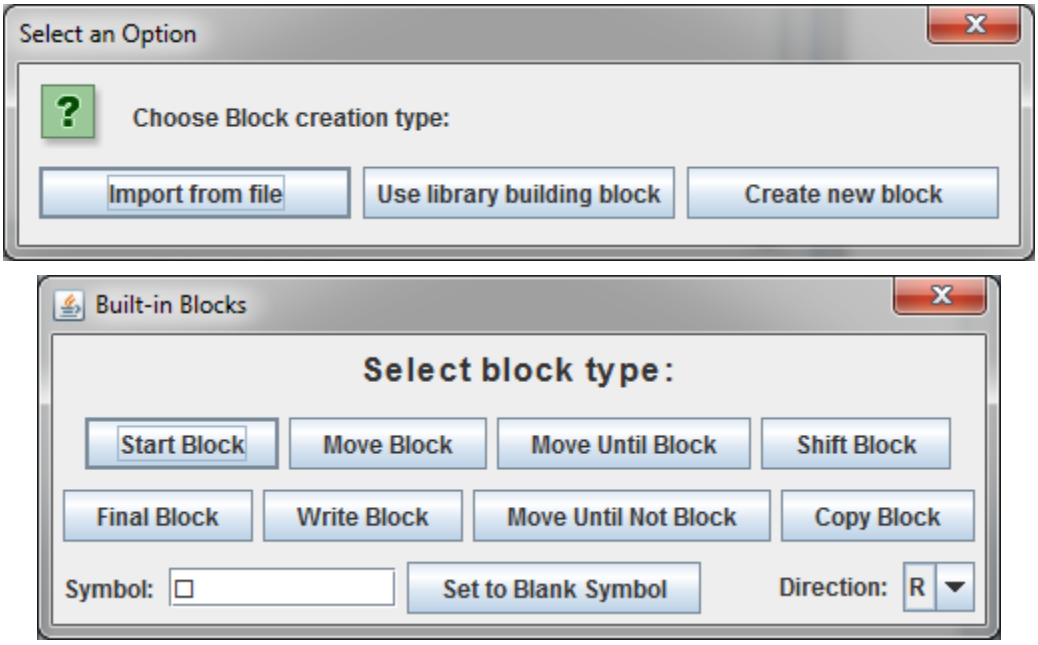


Figure 21: Block creation and Library building block selection menus in JFLAP v8.0

Similar to the Block TM library, the idea of a Universal Turing machine was one that the previous versions of JFLAP included as a file, but never fully explored. By designing algorithms and TMs to convert a Turing machine and its input into the unary encoding necessary to be simulated on a Universal Turing machine, as well as decode the unary output of the Universal simulation, we are able to see the full power of the Universal Turing machine. Though there is no GUI yet to show how these algorithms work, the underlying algorithms are now in place to show users how Universal Turing machines work or at least improve upon its use.

Also, we have an updated algorithm and graphic interface for converting TMs to Unrestricted Grammars (which was one of the more bug-ridden aspects of JFLAP v7.0 and benefits the most with the new alphabet capabilities) and removing transitions with a Stay movement (i.e. not

Right or Left) from single tape TMs, as seen in figure 22. There remains plenty to do with Turing machines, including some of what we've already started on, but we certainly have an excellent base to build upon.

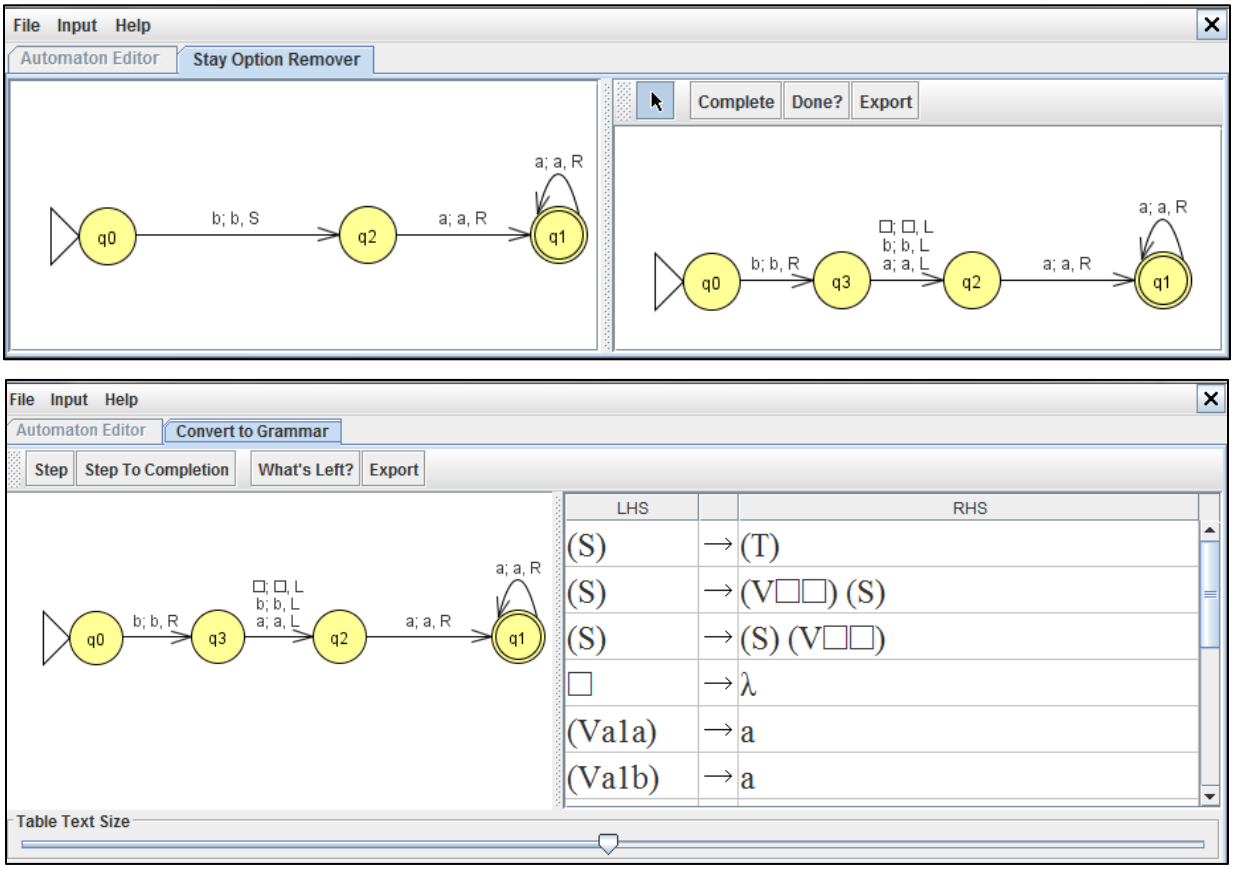


Figure 22: Stay option removal (top) and conversion to unrestricted grammar (bottom) for a single-tape TM.

## 7.4 Loading of Previous Safe Files

As the new codebase has increased the complexity of our objects, specifically the addition of complex symbols, alphabets, and formal definitions, the old XML format of saving files was not sufficient. We have created a new XML file format that encompasses items from before, but also includes new items (the new file extension is .jflap). The new format will not be detailed here, but as we want our new version to be just as capable as the old, it was decided that files created with the v7.0 (.jff) should be able to be read as well (though saving is only done in the new format). This means that all files a user may have created in the past, including all the sample files found on www.jflap.org, are available to use in the new version. There is one exception.

Since Turing machines with building blocks were designed differently in v8.0, one cannot read a TM with building blocks created from v7.0. This will be addressed in the future. Any old TM files using Blocks are not able to be read in, but will be addressed in the future.

# 8. Assessment Survey and Results

To judge our new version's effectiveness with the capabilities outlined in the previous sections, a small scale test was performed with students in Professor Rodger's Mathematical Foundations of Computer Science class. The survey itself along with the results can be found in appendix A of this document.

Overall, the results of this survey were very positive. Of the eight students present, seven participated in the assessment. Of the seven participants, six of them completed the entire assessment, with one completing a partial assessment. In order to analyze the anonymous responses in aggregate, each option was assigned a numeric score:

Strongly Agree = +2

Agree = +1

Neutral = 0

Disagree = -1

Strongly Disagree -2

Of the 13 scalable assessment questions, 12 of them yielded an average score greater than or equal to 1, meaning that the average student would tend to agree or strongly agree that JFLAP v8.0 had accomplished the stated task. Not a single participant disagreed or strongly disagreed with any of the statements, and all questions had a positive average value. The highest average assessment came with the statement that the new version of JFLAP increases flexibility with undo/redo capabilities in the automata GUI, with all students who answered strongly agreeing.

Of the non-scalable feedback questions, many participants further mentioned the undo/redo capabilities as the best addition, though some mentioned navigation, CYK parsing, and Language Generation. For the most part, negative feedback only came in the form of the Language Generator's freezing of the program (which occurs due to its exponential runtime), which as many mentioned can be dealt with by using multi-threaded programming.

# 9. Conclusions and Future Work

We believe that the addition of the new alphabets is one of the most sizable improvements to JFLAP in a long time due to its effect on the majority of the code. It opens up JFLAP's flexibility immensely, allowing users to generate real languages and sentences, include complex symbols, and contain bugs and limitations that arise due to directly manipulating Strings. Our various contributions, in their different levels of completion, will help progress JFLAP and keep it on the cutting edge of tools for visualizing these complex topics. With additions such as the Universal TM conversion/simulation/decoding, language generation, and possibly more complex L-Systems, JFLAP can provide a medium for work in the field of theoretical computer science that has not been present before. Going forward, we aim to focus on finalizing the graphic aspects of the remaining algorithms, including parsers, as well as expand upon what we've implemented already, and hope to incorporate what we've learned from software visualization research to release JFLAP v8.0 to the public.

# References

[BCS 1999] M. Byrne, R. Catrambone, and J. Stasko, Evaluating animations as student aids in learning computer algorithms. *Computers & Education,* Vol . 33, No. 4, pages 253-278, December 1999.

[BrH 1998] M. Brown and J. Hershberger. Fundamental Techniques for Algorithm Animation Displays. In *Software Visualization,* pages 81-101. Massachusetts Institute of Technology, Massachusetts, 1998.

[CSK 2011] P. Chakraborty, P. C. Saxena, and C. P. Katti, Fifty Years of Automata Simulation: A Review, ACM Inroads, Vol. 2, No. 4, pages 59-70, December 2011.

[CoS 1970] J. Cocke and J. T. Schwartz, Programming Languages and Their Compilers: Preliminary Notes, Second Revised Version, Courant Institute of Mathematical Sciences, New York, 1970.

[Gen 2012] J. Genkins, Improving the Flexibility of JFLAP: String Symbols, Explicit Formal Definitions, and Code Reconstruction, B.A. Thesis. Duke University. 2012.

[HDS 2002] C. Hundhausen, S. Douglas, and J. Stasko, A Meta-Study of Algorithm Visualization Effectiveness, *Journal of Visual Languages and Computing,* Vol. 13, No. 3, pages 259-290, June 2002.

[Lin 2006] Peter Linz. An Introduction to Formal Languages and Automata. Sudbury, MA: Jones and Bartlett, 2006.

[Kas 1965] T. Kasami, "An Efficient Recognition and Syntax Algorithm for Context-free Languages," Scientific Report AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, MA, 1965.

[Nap 2002] T. Naps (co-chair), G. Rossling (co-chair), V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger and J. Angel Velazquez-Iturbide, Exploring the Role of Visualization and Engagement in Computer Science Education, Report of the Working Group on "Improving the Educational Impact of Algorithm Visualization", SIGCSE Bulletin, Vol. 35, No. 2, p. 131-152, June 2003 (from ITiCSE 2002, Aarhus, Denmark, June 2002.)

[PBS, 1993] B. Price, R. Baecker, and I. Small, A Principled Taxonomy of Software Visualization. *Journal of Visual Languages and Computing*, Vol. 4, No. 3, pages 211-266. Sept. 1993.

[PrL 1990] P. Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants.* Springer-Verlag New York Inc., New York, 1990.

[Rod 2013] S. H. Rodger, JFLAP web site, www.jflap.org, accessed April 30, 2014.

[Ros 2008] R. J. Ross. Hypertextbooks and a Hypertextbook authoring environment. In Proceedings of ITiCSE'2008. pages 133-137

[RoF 2006] S. H. Rodger and T. W. Finley, JFLAP - An Interactive Formal Languages and Automata Package. Jones and Bartlett, Sudbury, MA, 2006.

[RQS 2011] S. H. Rodger, H. Qin, and J. Su, Increasing the use of JFLAP in Courses, Sixth Program Visualization Workshop (PVW 2011), Darmstadt, Germany, p. 53-56, 2011.

[SCH 1994] M. Stallman, R. Cleaveland, and P. Hebbar. Gdr: A visualization tool for graph algorithms. In Proceedings of Computational Support for Discrete Mathematics, pages 17-28. American Mathematical Society, 1994.

[SBL 1993] J. Stasko, A. Badre, and C. Lewis, Do algorithm animations assist learning?: an empirical study and analysis. In Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems. ACM, New York, NY, USA, 61-66. 1993.

[SSMN 2004] P. Saraiya, C. Shaffer, D. McCrickard, and C. North. Effective features of algorithm visualizations. In Proceedings of the 35th SIGCSE technical symposium on Computer science education. ACM, New York, NY, USA, 382-386. 2004.

[Sta 1998] J. Stasko. Smooth, Continuous Animation for Portraying Algorithms and Processes. In S*oftware Visualization*, pages 103-118. Massachusetts Institute of Technology, Massachusetts, 1998.

[Tay 1998] R. Taylor. Models of Computation and Formal Languages. Oxford University Press, New York, 1998.

[You 1967] D. H. Younger, "Recognition and Parsing of Context-Free Languages in Time $O(n^3)$" Information and Control, 10(2), pages 189-208, 1967.

# Appendix

# Appendix A. Survey

JFLAP v8.0 Assessment Survey
April 4, 2014

1. Language Generation

**a. Open the file named *language_gen.jflap***

**b. Write 3 strings in the language and 3 strings not in the language**

**c. Write a description of the language**

**d. Open the Language Generator**

Menu: "Input" → "Generate Language"

**e. Generate strings in the language using the text field**

- Enter a number *x* ("3" for example) into the text field
- Press the "# of Strings" button to generate the first *x* strings in the language
- Press the "String Length" button to generate all strings of length *x*
- Experiment with different values of *x*, though it is recommended you enter less than 14 for string length and less than 50 for "# of strings" to avoid freezing.

**f. Observation:**

Take a moment to study the strings, does it match your description of the language? Try entering other lengths or # of strings. Can you come up with a better description?

## Complete Assessment Question 1 Now

Assessment questions can be found on page 3 of this survey.

2. CYK Algorithm

**a. Open the file named *grammar_for_CYK.jflap***

**b. Here are 6 strings in the language:**

| | | |
|---|---|---|
| *b* | *abb* | *babbabb* |
| *bbbabb* | *abbbbb* | *babb* |

**Here are 3 strings NOT in the language:**

| | | |
|---|---|---|
| *abab* | *aaaabbbb* | *bbababb* |

**c. Open the CYK parser**

Menu: "Input" → "CYK Parse"

**d. Parse the string *babbabb* using the parser**

- Enter the string into the "Input" field and press the "Set" button.
- For the first row, fill in at least one box with all Variables that can derive the Terminal that is highlighted when that box is selected. For example, under any *a* Terminal, you enter the only Variable that can derive it, which is *A.*
- Select one box you haven't filled in and press the "Do Selected" button for an example of how the box should be filled. Once you have filled in the entire first row, or pressed the "Step" button, the next row will become active to be edited. Any incorrect entries will be marked in red.
- Select one of the boxes in the new row and press the "Animate Selected" button. Two boxes from the previous row will light up. This pair of boxes represents the RHS of a production. The leftmost box will represent the set of

possibilities for the first variable and the rightmost box the second. Enter all variables that can derive some combination of the two sets. For example, if the leftmost box is the set {*A*} and the rightmost box is the set {*C, S*}, you would enter all Variables that could derive *AC* or *AS*. To enter an empty set, edit the box, but leave it empty (or press the *space* or *backspace* keys).

- Complete the remainder of the table through manually entering variables or using any of the "Step", "Do Selected", and "Complete" buttons. If you get stuck, use the "Animate Selected" button, which will now highlight multiple pairs of boxes similar to before. This follows the same logic, there are just more possibilities to consider. Each pair of boxes is considered as before, and the final entry should contain all Variables that can derive any of the pairs.
- The input string will be accepted (or rejected) when the final box is filled in. If the Start Variable is contained in that set, the string is in the language, otherwise it is not.
- You may enter any of the strings above (or string you come up with) to test them against the parser, and (if accepted) can choose "Derivation View" from the dropdown menu to view the Derivation Tree and Table.

## Complete Assessment Question 2 Now

Assessment questions can be found on page 3 of this survey.

3. Suppose you have a language over the alphabet {*a, b*}, which includes all strings containing 0 or more *a*'s followed by an odd number of *b*'s.

a. Write 3 strings in the language and 3 strings not in the language

b. Construct a DFA for this language
"Main Menu" → "Finite Automaton"
In your construction, test some of the following features:

- Moving states with the state tool (hotkey s, use the right mouse key to drag and drop created states)
- Undo/Redo on features such as moving, creating, renaming, labeling, or deleting states, transitions, or notes (hotkeys ctrl-z/ctrl-y)
- Manipulating and deleting multiple objects (select by dragging or ctrl- /shift-clicking with arrow tool, delete with the "Delete" key)

## Complete Assessment Question 3 Now

Assessment questions can be found on page 3 of this survey.

Fill in the information below about your experience with JFLAP v8.0. If you cannot answer or feel that the statement does not apply to the exercise, select "N/A."

1. **Language Generation**

| | Strongly agree | Agree | Neutral | Disagree | Strongly Disagree | N/A |
|---|---|---|---|---|---|---|
| **The "# of Strings" function helps to understand languages.** | ○ | ○ | ○ | ○ | ○ | ○ |
| **The "String Length" function helps to understand languages.** | ○ | ○ | ○ | ○ | ○ | ○ |
| **The Language Generator makes it easier to explore grammars.** | ○ | ○ | ○ | ○ | ○ | ○ |
| **The Language Generator improves the capabilities of JFLAP as a teaching tool.** | ○ | ○ | ○ | ○ | ○ | ○ |

2. **CYK Algorithm**

| The new verison of JFLAP… | Strongly agree | Agree | Neutral | Disagree | Strongly Disagree | N/A |
|---|---|---|---|---|---|---|
| **… increases understanding of the CYK algorithm.** | ○ | ○ | ○ | ○ | ○ | ○ |
| **… provides useful feedback in filling out the CYK table.** | ○ | ○ | ○ | ○ | ○ | ○ |
| **… provides appropriate support with "Step," "Do Selected," and "Complete" functionality in the CYK table.** | ○ | ○ | ○ | ○ | ○ | ○ |
| **… provides appropriate assistance with "Highlight Selected" functionality in the CYK table.** | ○ | ○ | ○ | ○ | ○ | ○ |
| **… improves the flexibility of parsing strings in a language.** | ○ | ○ | ○ | ○ | ○ | ○ |

3. **Automata GUI**

| The new verison of JFLAP… | Strongly agree | Agree | Neutral | Disagree | Strongly Disagree | N/A |
|---|---|---|---|---|---|---|
| **… makes the formal definition of automata very clear.** | ○ | ○ | ○ | ○ | ○ | ○ |
| **… provides useful tools for manipulating automata.** | ○ | ○ | ○ | ○ | ○ | ○ |
| **… increases flexibility with undo/redo capabilities.** | ○ | ○ | ○ | ○ | ○ | ○ |
| **… allows for a more user-friendly Automata GUI** | ○ | ○ | ○ | ○ | ○ | ○ |

4. Of what you saw in JFLAP v8.0, what addition(s) do you like the most? Explain.

5. What do you like the least? Explain.

6. Provide any other feedback – Features you would like to see added to JFLAP, helpfulness of the interactions with the formal definition, the value error/warning messages, etc.

# Appendix B. Results

## Individual Results

The table below shows the individual participants responses to the survey and assessments of JFLAP v8.0. As discussed in section 8 of the paper, each category was given an associated value, and those values are what were input into this table. A Strongly Agree was assigned a value of +2, Agree +1, Neutral 0, Disagree -1, and Strongly Disagree -2. If the cell has an X, that means that the participant either did not choose to answer the question or selected the N/A option.

| Statement: | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| **The "# of Strings" function helps to understand languages.** | **1** | **2** | **1** | **2** | **1** | **0** | **2** |
| **The "String Length" function helps to understand languages.** | **1** | **2** | **0** | **2** | **0** | **0** | **2** |
| **The Language Generator makes it easier to explore grammars.** | **2** | **2** | **2** | **2** | **1** | **0** | **2** |
| **The Language Generator improves the capabilities of JFLAP as a teaching tool.** | **2** | **2** | **2** | **2** | **1** | **0** | **2** |
| **The new version of JFLAP increases understanding of the CYK algorithm.** | **1** | **1** | **1** | **1** | **2** | **X** | **1** |
| **The new version of JFLAP provides useful feedback in filling out the CYK table.** | **1** | **1** | **1** | **1** | **1** | **X** | **0** |
| **The new version of JFLAP provides appropriate support with "Step," "Do Selected," and "Complete" functionality in the CYK table.** | **0** | **1** | **2** | **1** | **0** | **X** | **2** |
| **The new version of JFLAP provides appropriate assistance with "Highlight Selected" functionality in the CYK table.** | **1** | **1** | **2** | **1** | **1** | **X** | **2** |
| **The new version of JFLAP improves the flexibility of parsing strings in a language.** | **2** | **1** | **2** | **1** | **1** | **X** | **2** |
| **The new version of JFLAP makes the formal definition of automata very clear.** | **2** | **1** | **1** | **2** | **1** | **X** | **2** |
| **The new version of JFLAP provides useful tools for manipulating automata.** | **1** | **1** | **1** | **2** | **1** | **X** | **2** |
| **The new version of JFLAP increases flexibility with undo/redo capabilities.** | **2** | **2** | **2** | **2** | **2** | **X** | **2** |
| **The new version of JFLAP allows for a more user-friendly Automata GUI** | **2** | **1** | **1** | **2** | **1** | **X** | **2** |

**Question 4: Of what you saw in JFLAP v8.0, what addition(s) do you like the most? Explain.**

2: I really liked the language generator, I think it greatly helps in adding a more visual step which makes easier to see the language description.

3: The undo/redo. That was a big help. I also thought the CYK feature was useful but I never explored that feature in v7.

5: I like the CYK parser a lot, and the undo/redo functions are awesome! Thank you!

6: "The language generator has potential, but it freezes far too fast much too quickly. It would be nice to get 1 random string of a random length rather than all strings of one length or list of the first # of strings in the grammar. Also, use multithreading to prevent freezing. ... ..."

7: The hotkeys are really handy, and I was able to use the language generator to check that my understanding of the grammar was correct.

**Question 5: What do you like the least? Explain.**

2: Nothing much, everything looks good

3: Entering things into the CYK table was a bit annoying because it is not immediately clear that you cannot separate the entires with a comma. Also, it is not obvious that you need to click 'next' before you move on.

5: The new version won't let me right-click on states in my automata to change them to final, initial, etc. I have a Mac, Mavericks OS, and I was the only person in the class with this issue so it might just be my computer, but I thought I'd bring it up.

6: "Freezing. The words set and change are misleading."

**Question 6: Provide any other feedback – Features you would like to see added to JFLAP, helpfulness of the interactions with the formal definition, the value error/warning messages, etc.**

3: Would it be possible to allow the parsing to happen on a separate thread so it does not freeze the program?

5: The Step, Complete, etc. were a little difficult to differentiate on their own. Maybe an additional label underneath them to describe each of their functions.

## Aggregate scores:

The table below shows the aggregate sum, mean score, and individual score counts for each assessment statement. Green highlighted averages indicate all averages greater than or equal to 1, i.e. every situation in which the average student agreed or strongly agreed with the statement. Note that the disagreed and strongly disagreed columns were removed as there was not a single instance of either selection.

| Statement: | Sum | Mean | Strongly Agree | Agree | Neutral |
|---|---|---|---|---|---|
| The "# of Strings" function helps to understand languages. | 9 | 1.29 | 3 | 3 | 1 |
| The "String Length" function helps to understand languages. | 7 | 1 | 3 | 1 | 3 |
| The Language Generator makes it easier to explore grammars. | 11 | 1.57 | 5 | 1 | 1 |
| The Language Generator improves the capabilities of JFLAP as a teaching tool. | 11 | 1.57 | 5 | 1 | 1 |
| The new version of JFLAP increases understanding of the CYK algorithm. | 7 | 1.16 | 1 | 5 | 0 |
| The new version of JFLAP provides useful feedback in filling out the CYK table. | 5 | 0.83 | 0 | 5 | 1 |
| The new version of JFLAP provides appropriate support with "Step," "Do Selected," and "Complete" functionality in the CYK table. | 6 | 1 | 2 | 2 | 2 |
| The new version of JFLAP provides appropriate assistance with "Highlight Selected" functionality in the CYK table. | 8 | 1.33 | 2 | 4 | 0 |
| The new version of JFLAP improves the flexibility of parsing strings in a language. | 9 | 1.5 | 3 | 3 | 0 |
| The new version of JFLAP makes the formal definition of automata very clear. | 9 | 1.5 | 3 | 3 | 0 |
| The new version of JFLAP provides useful tools for manipulating automata. | 8 | 1.33 | 2 | 4 | 0 |
| The new version of JFLAP increases flexibility with undo/redo capabilities. | 12 | 2 | 6 | 0 | 0 |
| The new version of JFLAP allows for a more user-friendly Automata GUI | 9 | 1.5 | 3 | 3 | 0 |